The q-Interface to UNIX

Robert D. Russell

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The q-interface to UNIX

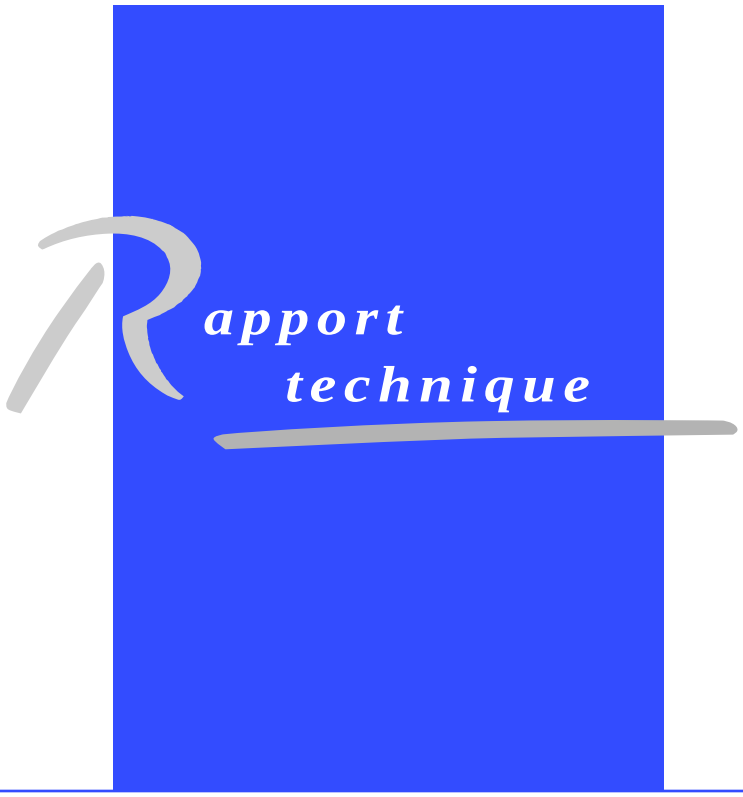
Robert D. Russell

LIP, ENS Lyon

No 0222

July 1998

THÈME 1



ISSN 0249-0803

RHÔNE-ALPES

The q-interface to UNIX

Robert D. Russell *
LIP, ENS Lyon

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport technique n ° 0222 — July 1998 — 23 pages

Abstract: This report describes an application program interface to UNIX called the q-interface. It is designed to provide the programmer with convenient access to a reliable, message-oriented asynchronous network transport service, and to asynchronous event handling in general. It utilizes a "call-back" paradigm to notify the programmer of the occurrence of asynchronous events, and guarantees that all call-backs are atomic. It hides both the notification mechanism of the underlying operating system and the underlying networking system, which in the first implementation is TCP/IP.

Key-words: Asynchronous I/O. Reliable Message-Oriented Transport Service.

(Résumé : tsvp)

* On leave during the 1997-98 academic year as an Associated Professor at the Laboratoire de l'Informatique du Parallélisme, École normale supérieure de Lyon. Permanent address : Department of Computer Science, Kingsbury Hall, University of New Hampshire, Durham, NH 03824-3591, USA. Email : rdr@unh.edu

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

L'interface de programmation UNIX "q-interface"

Résumé : Ce rapport décrit une interface de programmation à UNIX nommée la q-interface. Cette interface a été conçue pour fournir au programmeur un accès facile à un service de transport fiable et asynchrone basé sur l'envoi de messages, et à un traitement asynchrone d'événements en général. Elle utilise le concept de fonctions "call-back" afin d'informer le programmeur de la présence d'événements asynchrones, et il garantit que tous les call-backs sont atomiques. Enfin, elle cache le mécanisme de notification du système d'exploitation sous-jacent ainsi que le protocole réseau sous-jacent, qui, dans notre première implémentation, est TCP/IP.

Mots-clé : E/S asynchrones. Envoi de messages. Service de transport fiable.

1 Introduction

This report describes an application program interface to UNIX called the q-interface. It is designed to provide the programmer with convenient access to asynchronous signals, to asynchronous device I/O, and to a reliable, message-oriented asynchronous network transport service. If the underlying operating system does not provide these features directly, the q-interface will simulate them as best it can. The name "q-interface" was chosen because all I/O requests are simply enqueued at the time the request is submitted to the interface, and are dequeued and executed asynchronously by the interface only when it determines that the I/O device is ready to perform the operation.

The q-interface utilizes a "call-back" paradigm to express asynchronous events. This paradigm is very different from the "normal" UNIX-style API, and is closer to that found with X-Windows programming. When establishing the conditions that may trigger an asynchronous event (such as an I/O operation that will complete at some future time), the programmer must specify to the q-interface a "call-back function" that the q-interface will invoke if and when the asynchronous event actually occurs.

The need for such an interface arises from two considerations : convenience and portability.

Most currently available versions of the UNIX operating system do not provide any way to perform asynchronous I/O or parallel operations (i.e., threads). Although the POSIX standard includes specifications for such features, they are not yet widely available. The q-interface provides a convenient way to simulate asynchronous operation. Even on systems that provide the POSIX asynchronous I/O features, users may find the q-interface "call-back" style of programming more convenient to use, since it guarantees atomic operation for each call-back routine with respect to other call-back functions (subject to certain restrictions, see Section 9). It also provides a "call-back" method for dealing with UNIX signals. This is very convenient for the programmer, since it usually eliminates the need for explicit synchronization between various functions and the need to mask and unmask events. It also makes portability easier, since if the POSIX features mentioned above should become available, it would be possible to reimplement the q-interface to use those features, in which case all programs utilizing the q-interface will automatically take advantage of the new features without reprogramming.

The q-interface also addresses convenience and portability for network interaction. It provides a reliable, message-oriented transport service, something which is missing in the normal TCP/IP protocol stack, and it hides the details of TCP and IP, making it possible to implement different versions of the q-interface for different networking protocol stacks without having to reprogram code utilizing the q-interface.

This report is organized as follows. Section 2 describes the functions used to initialize and terminate the q-interface. This is followed in Section 3 by a description of the general buffering mechanism utilized throughout the q-interface, and in Section 4 by a description of the general call-back mechanism utilized throughout the q-interface. Section 5 describes the reliable, client-server message transport service. Section 6 describes the functions for performing and canceling asynchronous I/O. This is followed in Section 7 by a description of functions for performing general asynchronous notification based on time, and in Section 8 by a description of functions to perform asynchronous signal handling. Section 9 describes how the q-interface simulates asynchronous operation. Finally, Section 10 describes some timing functions that are useful for debugging and performance evaluation. Section A.1 of the Appendix contains a sample program illustrating the use of some q-interface functions, and Section A.2 gives an alphabetical listing of the prototypes of all the functions in the q-interface.

2 Initialization Functions

In order to initialize the q-interface, the function **q_setup** must be called before any other functions. By analogy, **q_endup** must be the last function called after the programmer has finished using all other functions. For programmer convenience in exiting a program, the **q_exit** function calls **q_endup** and then calls **exit**. The function **q_reset** is called by a newly-spawned child process to reset the q-interface set up by and therefore inherited from the parent process. Since this inherited version contains a copy of everything in the parent's version, **q_reset** frees all enqueued requests and resets all enqueued signals to their default actions.

```
extern void q_setup( unsigned int pre_size,
        unsigned int min_size,
        unsigned int max_size );

extern void q_endup( void );

extern void q_exit( int status );

extern void q_reset( void );
```

As part of the q-interface initialization, the programmer must define the general form of the message buffers to be used for all network I/O. Details about these buffers are given in the next section. The three input parameters to **q_setup** specify, respectively,

- **pre_size** : the size of the message preamble. This size must be greater than or equal to **sizeof(void *)**, and also a multiple of **sizeof(void *)**. The preamble part of a buffer never appears on the network — it is used to store internal information, such as pointers that link a buffer onto a list. The first **sizeof(void *)** bytes of the preamble are reserved for use by the q-interface.
- **min_size** : the size of the minimum message sent/received on the network, which is the same as the fixed size of the message header. This size must be greater than or equal to 8, and also a multiple of 8. This size excludes the preamble size. The first 4 bytes of the header are reserved for use by the q-interface.
- **max_size** : the size of the maximum message sent/received on the network. This size must be greater than or equal to **min_size**, and also a multiple of 8. This size includes the header size but excludes the preamble size.

When buffers are allocated as part of the q-interface's operation, they will occupy **pre_size** + **max_size** bytes of memory. The relationships between the parameters to **q_setup** and the parts of a buffer are shown in Figure 1.

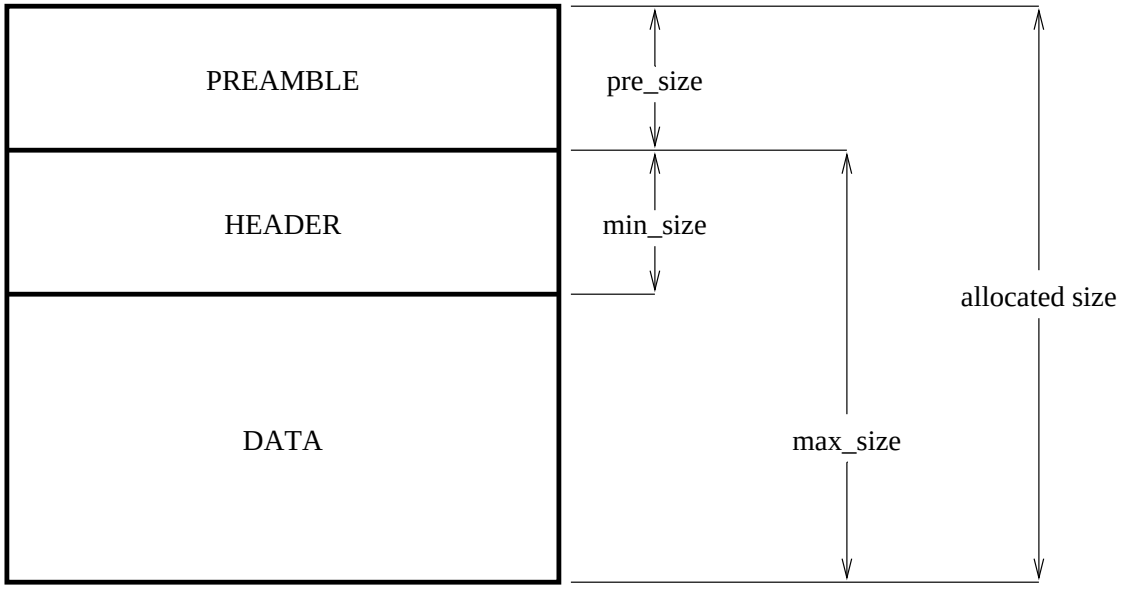


FIG. 1: q-interface buffer structure and sizes.

3 Buffering

An essential data structure used by the q-interface is the buffer structure shown in Figure 1. A buffer structure consists of a contiguous block of memory, and is subdivided into three parts : a "preamble", a "header", and a "data part". Except for the bytes reserved at the front of the preamble and at the front of the header, the size and internal format of each part are of no concern to the q-interface — it is up to the user of the q-interface to define these as he wishes. The preamble contains information that is not transmitted on the network. For example, the preamble might contain pointers used to link a buffer onto a list. The first **sizeof(void *)** bytes of the preamble are reserved for use by the q-interface itself. The header contains the "fixed" part of every message transmitted on the network, and as such is the minimum sized message that can be sent or received. The first 4 bytes of the header are reserved for use by the q-interface itself (in order to determine message boundaries within an underlying byte stream). The data part contains the "variable" part of every message transmitted on the network. The size of the preamble and header, and the maximum size of the data part, must be specified as part of the q-interface initialization, as explained above and shown in Figure 1.

Buffer structures are allocated and released dynamically by calling the following functions :

```
extern char *get_a_buffer( void );

extern char *get_a_header( const char *data_ptr );

extern char *dup_a_buffer(
        const char *oldbuf,
        unsigned int length );

extern void free_a_buffer( char *ptr );

extern void free_a_header( char *ptr );
```

get_a_buffer allocates a new, maximum size buffer, stores a NULL in the first **sizeof(void *)** bytes, and returns a pointer to it. Except for the first **sizeof(void *)** bytes, the value of all other bytes in the buffer are undefined.

get_a_header allocates a new, minimum size buffer (i.e., no data part), stores a copy of the input parameter **data_ptr** into the first **sizeof(void *)** bytes, and returns a pointer to it. The value of **data_ptr** must not be NULL. Except for the first **sizeof(void *)** bytes, the value of all other bytes in the buffer are undefined.

dup_a_buffer allocates a new, maximum size buffer, stores a NULL in the first **sizeof(void *)** bytes, copies into its header and data part **length** bytes from the header and data part of the buffer pointed to by the input parameter **oldbuf**, and then returns a pointer to it. Except for the first **sizeof(void *)** bytes, the value of all other bytes in the preamble are undefined.

The first **sizeof(void *)** bytes of any buffer allocated by the functions **get_a_buffer**, **get_a_header** or **dup_a_buffer** must not be modified by the calling program.

free_a_buffer deallocates the buffer pointed to by the input parameter **ptr**. This can be either a maximum size or minimum size buffer.

free_a_header deallocates the minimum size buffer pointed to by the input parameter **ptr**.

There is one other function in the q-interface that deals with buffers. Its prototype is :

```
extern void dump_buffer_stats( void );
```

This function simply prints to **stderr** a table of statistics about buffer utilization. Most of the numbers are usage counts accumulated since the interface was first initialized by the call to **q_setup**. This information is useful during debugging and performance tuning in order to evaluate how buffers are being utilized.

4 The Call-Back Mechanism

In order to provide asynchronous operation, the q-interface utilizes a "call-back" mechanism similar to that found in X-Windows. Many functions, such as those that make network connections and perform network I/O, set up call-backs by specifying the address of a function that conforms to the following prototype :

```
void call_back( int fd, char *buf, unsigned int len );
```

When invoked asynchronously by the q-interface, **call_back** accepts three input parameters :

- **int fd**, which will contain the I/O "handle" for any call-back associated with an I/O request to that handle.
- **char *buf**, which will contain a pointer to the I/O buffer for any call-back associated with an I/O request.
- **unsigned int len**, which will contain the actual length in bytes of an I/O operation for any call-back associated with an I/O request.

For call-backs not associated with I/O operations, the values supplied by the q-interface for each of the three parameters are described in the section where the function setting up the call-back is defined.

Note that many functions in the q-interface require the caller to supply a parameter containing the address of a call-back function to be called when an asynchronous event associated with the function occurs. If the user does not wish to receive a call-back when the indicated asynchronous event occurs, simply supply a NULL as the value of the call-back parameter.

5 Client-Server Message Transport

The q-interface provides a reliable, message-oriented network transport service based on the client-server model. Network addresses are represented by the abstract notion of a "location". A server process must first establish itself as a server at a certain location by registering itself at that location. Once this has been done, client processes can create connections to the server processes at any time, and the server process will be notified of each new connection by the call-back mechanism. Once a connection has been created, messages can flow in either direction. Each message is sent or received in a unique buffer. When sending a message, the sending process specifies a call-back function that is invoked asynchronously when the message has been sent and the buffer containing it can be freed. In order to receive a message, the receiving process specifies a buffer and a call-back function that will be invoked asynchronously when a new message has been received into the buffer. Appendix A.1 illustrates a simple client-server application that uses the q-interface.

5.1 Network Locations

There are two aspects to a "location" — its internal representation and its external representation — and the contents of each depends on the implementation of the q-interface.

The internal representation is an opaque value occupying 8 bytes, which must be in network byte order, since locations may themselves sent over the network between heterogeneous nodes. The external representation is an ASCII equivalent that can be represented in memory as a C string and stored in a file as text. Note that there is no automatic interoperability between different network representations in different implementations of the q-interface.

In TCP/IP, for example, a "location" consists of an IP address and port number. In the internal representation, the opaque value will contain the IP address as a 32-bit binary number and the port number as a 16-bit binary number, both in network byte order. In the external representation, the location will be the string representation of an IP address in "dotted-decimal" notation or an equivalent DNS name, followed by a colon followed by the port number as an integer. The port number can be omitted from the external representation, in which case a "well known port number" is used by default when the external representation is converted into internal form.

```
extern int q_strtolocation( const char *name,
        const char *default_name,
        q_location_t *location );

extern int q_locationtostr( q_location_t *location,
        unsigned int namelen,
        char *name );

extern char *q_strlocation( q_location_t *location );
```

These functions allow a program to convert between the internal and external forms of a network location.

q_strtolocation converts the external representation of a location given in the input parameter **name** into an equivalent internal representation stored in the output parameter **location**. The additional input parameter, **default_name**, is used to supply any components missing from **name** that are necessary for the formation of **location**. For example, in TCP/IP, a "well known port number" supplied in the **default_name** parameter would be used if there were no explicit port number specified in **name**. Both **name** and **default_name** are null-terminated C strings, and **location** is an opaque 8-byte structure containing the internal representation of a location. **q_strtolocation** returns the value 0 if it was able to perform the conversion, -1 if not, in which case the value in the output parameter is undefined.

q_locationtostr converts the internal representation of a location given in the **location** input parameter into an equivalent external representation that it stores as a null-terminated C string in the output parameter **name**. The storage for this string must be allocated by the caller, and must contain **namelen** bytes. The return value is the length of the string actually stored in **name**, excluding the null terminator. If this value is positive, it is the same as strlen(**name**) and will never be larger than **namelen** − 1. If this value is negative, it means that **namelen** was not big enough to hold the entire name, and its absolute value is the number of additional bytes needed to store the entire name as a null-terminated string. In this case, the first **namelen** − 1 bytes of the name will be stored in **name**, followed by the null-terminator.

q_strlocation converts the internal representation of a location given in the **location** input parameter into an equivalent external representation that it stores as a null-terminated C string in a fixed system-defined area that will be overwritten on each call. The value returned by **q_strlocation** is a pointer to this location. This function is provided primarily as a convenience for use during debugging.

5.2 Opening a Network Server Connection

A server process must go through two steps in order to connect to a client :

1. it must create a "listening post" at a location that it will advertize to clients ;
2. it must set up a call-back function that will be invoked asynchronously whenever a client actually makes a connection.

5.2.1 Establishing a Server Listening Post

```
extern int q_openlisteningpost( q_location_t *server_request,
        q_location_t *server_addr );

extern int q_openchildlisteningpost(
        q_location_t *server_request,
        q_location_t *server_addr );
```

These two functions create a server "listening post" by which a process can accept connections from remote client processes. The input parameter **server_request** is the listening post address by which the programmer wants the server process to be known to clients. If the output parameter **server_addr** is not NULL, it must point to a location into which these functions will copy the actual address assigned by the system to the server process. If these functions perform without error they will return a non-negative integer value that is the "handle" to be used in all other functions referencing this listening post. (In UNIX terminology, a "handle" is usually called a "file descriptor".) Note that upon return this "handle" is NOT

connected to any client process — it has simply been set up to accept future connections. If an error is detected, these functions will return a negative value whose absolute value is a system-defined error code.

The difference between these two functions is that **q_openlisteningpost** will utilize the entire listening post address specified in the input parameter **server_request**, whereas **q_openchildlisteningpost** will only utilize that part of the listening post address that specifies the interface on the host computer. In terms of IP addresses, this allows the user to specify a "well-known port" when **q_openlisteningpost** is used, but always forces the system to supply an unused port number when **q_openchildlisteningpost** is used. In addition, if the user supplies a NULL value for the parameter **server_request**, **q_openlisteningpost** will listen on all available interfaces on a multi-homed host, whereas **q_openchildlisteningpost** will only listen on the principle interface of the host.

5.2.2 Accepting a Server Connection from a Remote Client

```
extern int q_acceptconnection( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int));
```

This function is called by a server after it has previously established a listening post handle, in order to accept connections from remote clients on that handle. The input parameter **fd** is the listening post handle, and the input parameter **call_back** is the name of a call-back function that is invoked asynchronously by the q-interface at the time a client actually makes a connection (by calling **q_openclient**). **q_acceptconnection** itself does not block, but returns immediately after enqueuing **call_back**. The function value returned by **q_acceptconnection** will be 0 if it was able to perform without error, -1 otherwise.

When invoked asynchronously by the q-interface, **call_back** accepts three input parameters :

- An **int** containing the new handle for the newly accepted connection to a remote client. Note this value is different from the handle **fd** supplied as the input parameter to **q_acceptconnection**.
- A **char *** containing the pointer which was the input parameter **buffer** to **q_acceptconnection**. If this pointer is not NULL, the buffer it points to will contain two locations filled in by the q-interface : one describing the client side of the new connection, the other describing the server side of the new connection.
- An **unsigned int** containing the value of the input parameter **maxlen** to **q_acceptconnection**.

Note that a server process only has to call **q_acceptconnection** once for each listening post handle **fd**, since that single call causes the q-interface to repeatedly accept client connections and make call-backs to **call_back** until the handle **fd** is closed. Since the same pointer **buffer** is supplied as a parameter on each call-back, the function **call_back** must make a copy of any of the data it needs to retain from that buffer before returning. Since the q-interface guarantees that all call-backs are atomic, the user does not have to worry about the data in **buffer** from one connection being overwritten by data from another connection during the execution of the call-back function itself.

5.3 Opening a Network Client Connection

```
extern int q_openclient( q_location_t *server_addr,
        q_location_t *client_addr );
```

This function creates a reliable, message-oriented network transport connection between the calling client process and the remote server process indicated by the input parameter **server_addr**. If the output parameter **client_addr** is not NULL, it must point to a location into which this function will copy the actual address assigned by the system to this process's side of the connection. If this function performs without error it will return a non-negative value that is the "handle" to be used in all other functions referencing this connection. If an error is detected, this function will return a negative value whose absolute value is a system-defined error code.

5.4 Closing Network Connections

```
extern int q_close( int fd );
```

This function allows a program to close a network connection or network listening post on the handle **fd**. The function value returned by **q_close** will be 0 if it was able to perform without error, -1 otherwise.

6 Asynchronous I/O Requests

These functions enable a program to send, receive and cancel messages on the handle **fd**. Although designed for use with handles for network connections, they will also work correctly with handles (i.e., UNIX file descriptors) connected to any UNIX I/O device or file, including pipes, provided that these handles have been set into non-blocking I/O mode if they are to be used with readblock and/or writeblock operations.

6.1 Asynchronous Write Requests

```
extern int q_writeblock( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int) );

extern int q_writeblocktimeout( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );

extern int q_writejustdata( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int) );

extern int q_writejustdatatimeout( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );
```

The **buffer** parameter to all these routines must point to a buffer structure as shown in Figure 1. The **q_writeblock** and **q_writeblocktimeout** functions write the header and data portions of that buffer, and for these two functions the **length** parameter includes the number of bytes in both the header and data portions. The two other functions, **q_writejustdata** and **q_writejustdatatimeout**, write just the data portions of that buffer, and therefore the **length** parameter gives only the number of bytes of data.

q_writeblock enqueues a write of **length** bytes from the header and data areas of the buffer pointed to by **buffer** to the destination connected to the handle **fd**. The value of the first 4 bytes of the header will be overwritten by the q-interface for its internal use. This function returns to its caller as soon as the write request has been enqueued. The value returned will be 0 if this enqueuing succeeds, -1 if any error is detected.

When all **length** bytes have actually been sent to the destination, the function **call_back** will be invoked asynchronously. The three input parameters to **call_back** will be the value of **fd**, the value of **buffer**, and the total number of bytes actually sent (which is always equal to the value of **length** unless an error occurred or the request was cancelled).

Note that the preamble part of the buffer is never written and its size is not included in **length**. If the first **sizeof(void *)** bytes of the preamble are not NULL, they are assumed to point to an area of memory that contains the data portion of the message, in which case the message actually written to the destination will

be composed of the header contained in **buffer** followed by the data pointed to by the first **sizeof(void *)** bytes of the preamble. In this case, the buffer pointed to by **buffer** should be a minimum size buffer allocated by **get_a_header**. If the first **sizeof(void *)** bytes of the preamble are NULL, the data portion of the message will be taken from the buffer itself. In both cases, the value of **length** must include the sum of the lengths of both the header and the data portion of the message.

The function **q_writeblocktimeout** is similar to **q_writeblock**, except that after writing is complete it will not invoke the function pointed to by **call_back** before the time represented by the input parameter **deadline**. There is no guarantee that the write itself will complete before the deadline — if it does, the call-back is delayed to the deadline; if it does not, the call-back will occur as soon as the write completes. This time is an absolute time in the same format as the wall-clock time returned by the UNIX function **gettimeofday**.

The function **q_writejustdata** (**q_writejustdatatimeout**) is similar to the function **q_writeblock** (**q_writeblocktimeout**)), except that only the data portion of the buffer is written and the **length** parameter should be set to just the number of bytes of data. Since no header is written, no part of the buffer is overwritten by the q-interface. However, no boundaries between messages written by these two functions are maintained, which allows data written by them to be read by ordinary UNIX read operations.

6.2 Asynchronous Read Requests

```
extern int q_readblock( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int) );

extern int q_readblocktimeout( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );

extern int q_readjustdata( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int) );

extern int q_readjustdatatimeout( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );
```

The **buffer** parameter to all these routines must point to a buffer structure as shown in Figure 1. The **q_readblock** and **q_readblocktimeout** functions read the header and data portions of that buffer, and for these two functions the **maxlen** parameter includes the number of bytes in both the header and data portions. The two other functions, **q_readjustdata** and **q_readjustdatatimeout**, read just the data portions of that buffer, and therefore the **maxlen** parameter gives only the maximum number of bytes of data expected.

q_readblock enqueues a read of a message of up to **maxlen** bytes into the header and data areas of the buffer pointed to by **buffer** from the source connected to the handle **fd**. The value of the first 4 bytes of the header will be undefined. This function returns to its caller as soon as the read request has been enqueued. The value returned will be 0 if this enqueuing succeeds, -1 if any error is detected.

When a complete message has been received from the source, the function **call_back** will be invoked asynchronously. The three input parameters to **call_back** will be the value of **fd**, the value of **buffer**, and the total number of bytes actually received (which is always less than or equal to the value of **maxlen**).

Note that the preamble part of the buffer is never overwritten and its size is not included in **length**. However, if the first **sizeof(void *)** bytes of the preamble are not NULL, they are assumed to point to an area of memory into which the data portion of the message will be read. In this case, the buffer pointed to by **buffer** should be a minimum size buffer allocated by **get_a_header**. If the first **sizeof(void *)** bytes of the preamble are NULL, the data portion of the message will be read into the buffer itself. In both cases, the number of bytes given in the call-back will be the sum of the lengths of both the header and the data portion of the message.

q_readblocktimeout is similar to **q_readblock**, except that after reading is complete it will not invoke the function pointed to by **call_back** before the time represented by the input parameter **deadline**. There is no guarantee that the read itself will complete before the deadline — if it does, the call-back is delayed to the deadline; if it does not, the call-back will occur as soon as the read completes. This time is an absolute time in the same format as the wall-clock time returned by the UNIX function **gettimeofday**.

The function **q_readjustdata** (**q_readjustdatatimeout**) is similar to the function **q_readblock** (**q_readblocktimeout**)), except that only the data portion of the buffer is read and the **maxlen** parameter should be set to just the maximum number of bytes of data expected. The header portion of the buffer will be undefined, since these two functions do not read a header. Without a header, the q-interface cannot ensure that the size of the message received is the same as the size of the message sent (i.e., message boundaries are not preserved by these functions). A message is considered complete as soon as at least one byte has been received. The actual number delivered in the buffer is all available bytes up to the maximum specified by **maxlen** (i.e., the request waits only until "at least one" byte is available, then takes all that it can find at that time). By not reading a header, these two functions make it possible to read data written by ordinary UNIX write operations, whereas **q_readblock** and **q_readblocktimeout** should be used only to read data written by **q_writeblock** or **q_writeblocktimeout**.

6.3 Canceling Outstanding I/O Requests

```
extern void q_cancel( int fd,
        int who,
        int do_call_backs );
```

This function cancels outstanding I/O requests enqueued on the handle **fd**. If the value of the input parameter **who** is 1, only read requests will be cancelled; if it is 2, only write requests will be cancelled; if it is 3, both read and write requests will be cancelled. If the value of the input parameter **do_call_backs** is non-zero, the call-back function associated with each cancelled request will be invoked by **q_cancel** with its total bytes parameter having the value 0. If **q_cancel** is itself called as part of a call-back function, then the atomicity of that call-back will be violated by each call-back for a cancelled request. If the value of **do_call_backs** is zero, the call-back function associated with each cancelled request will not be invoked by **q_cancel**.

7 Asynchronous Timed Requests

```
extern int q_deadlinetimeout( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );
```

```
extern int q_intervaltimeout( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *interval );
```

These functions allow a program to receive an asynchronous event at a specified time. Both functions return immediately after enqueuing the call-back function for the event. Their return value will be 0 if everything is ok, -1 if any error is detected.

The difference between these two functions is that **q_deadlinetimeout** takes as an input parameter an absolute "deadline" time, whereas the function **q_intervaltimeout** takes a relative "interval" time. Internally, the q-interface will compute a deadline from this interval by adding to it the current wall-clock time returned by the UNIX function **gettimeofday**.

When the current wall-clock time becomes greater than or equal to the deadline, the function **call_back** will be invoked asynchronously, with the values of **fd**, **buffer**, and **length** supplied as input parameters. These values are not otherwise used by either **q_deadlinetimeout** or **q_intervaltimeout**.

8 Asynchronous Signal Handling

```
extern int q_signal( int signo,
        char *buffer,
        unsigned int len,
        void (*call_back)(int, char *, unsigned int) );
```

This function allows a program to receive an asynchronous call-back whenever the signal indicated by **signo** occurs. **q_signal** itself enqueues the request and sets up a signal handler within the q-interface before returning immediately to the caller. The value returned is 0 if no error is detected, -1 otherwise. Note that the signal **signo** is neither masked nor unmasked by **q_signal**. If **signo** was enqueued by a prior call to **q_signal**, this call will simply overwrite the previous enqueuing. If the value of the **call_back** parameter is NULL, any previous enqueuing for **signo** is removed and handling for this signal reverts to the system-defined default (SIG_DFL).

q_signal needs to be called only once for each unique **signo**, since that single call causes the q-interface to repeatedly make asynchronous invocations of the function **call_back** whenever the indicated signal occurs. The three parameters passed to **call_back** are the signal number **signo**, the value of **buffer** and the value of **len**. Except for this usage, no other reference is made to either **buffer** or **len** by the q-interface. The q-interface guarantees that all call-backs are atomic, so that the execution of **call_back** due to one signal occurrence will not be interrupted by any another signal occurrence. Of course, UNIX signals are an unreliable notification mechanism, and the q-interface makes no attempt to correct this defect.

9 Simulating Asynchronous Operation

```
extern void q_loop( void );

extern void q_wait( struct timeval *timeout );

extern void q_poll( void );

extern void q_block( void );

extern void q_probe( int fd );
```

These functions allow a program to simulate the asynchronous handling of events and I/O operations. They cause the q-interface to actually dequeue requests and invoke call-back functions if the corresponding

asynchronous events have occurred. They should not normally be called from within call-back functions themselves, as doing so will violate the atomic property of call-back functions.

Normally the only function utilized by a "passive server" program is **q_loop**, which never returns to its caller. **q_loop** causes control to transfer to and remain in the q-interface which waits to satisfy enqueued requests until some asynchronous event occurs, at which time the q-interface will invoke the call-back function enqueued to deal with that event. When that call-back function returns, control returns to the q-interface to continue looking for requests to satisfy and to await the next event. Therefore, **q_loop** should be called only after all call-back functions for such events have been set up by previous calls to the appropriate q-interface function. Typically in a "server" program, these previous calls would be to **q_openlisteningpost** and **q_acceptconnection** in order to establish the program as a server. Once **q_loop** has been called, whenever a new client makes a connection, the q-interface will invoke the call-back function set up by **q_acceptconnection**, and that function in turn would set up call-backs appropriate for the I/O requests between the server and the new client. See Appendix A.1.2.

The other functions listed above are normally utilized by "client" programs, not "server" programs, because clients are "active", not "passive" in their use of network and other I/O devices. Clients will generally not use **q_loop**, since they will need to retain control in order to perform their computations. However, they will use q-interface functions to set up network connections and to perform network I/O, for example, and will therefore have to indicate to the q-interface the appropriate synchronization points in their operation at which the q-interface may test to see if requests can be satisfied and call-backs invoked.

q_wait takes as its input parameter **timeout** a pointer to a time interval structure, and will cause the q-interface to wait until either at least one asynchronous event of any kind occurs or the amount of time indicated by **timeout** has elapsed before returning to its caller.

q_poll is equivalent to **q_wait** with **timeout** pointing to a structure specifying a time interval of zero. Therefore, it simply invokes any call-backs that are ready and then returns immediately to its caller.

q_block is equivalent to **q_wait** with a **timeout** value of NULL, indicating an "infinite" interval of time. Therefore, it waits indefinitely until at least one asynchronous event occurs before returning to its caller. Note that calling **q_loop** is equivalent to calling **q_block** inside an infinite loop.

q_probe takes as its input parameter **fd** an I/O handle which it uses to probe the associated I/O device to see if it is ready to perform an enqueued request. If it is, the I/O operation is performed and the corresponding call-back function invoked when it completes. **q_probe** returns to its caller after any requests are performed or call-backs are invoked. If no requests are enqueued or are ready to be performed, **q_probe** returns immediately. Because its operation is only a subset of the operation of **q_poll**, **q_probe** is normally used only when the caller has reason to believe that the indicated I/O device has an enqueued request that might be ready to perform (perhaps because the q-interface function to enqueue the request was just called).

10 Timing Functions

```
extern void q_start_stamp( int fd,
        int mode,
        unsigned int max_stamp );

extern void q_stop_stamp( unsigned int max_stamp_print );

extern void dump_queue_stats( void );
```

These functions allow a program to record the start and stop times around the code that reads or writes to an I/O device, and to dump these results.

q_start_stamp is called to start recording time stamps for I/O requests to the device indicated by the handle **fd**. Only one device at a time can perform such recording. If the input parameter **mode** is zero, read requests will be recorded ; if it is non-zero, write requests will be recorded. The input parameter **max_stamp** indicates the maximum number of time stamps to record. Each request requires two time stamps : one recorded before the first attempt to satisfy the request, the second recorded after the request is completely

satisfied (but before the call-back function is invoked). Therefore, timings for at most **max_stamp** ÷ 2 requests will be recorded. Note that under "normal" operation, when **q_start_stamp** is not called, the q-interface will not record timings for any requests.

q_stop_stamp is called to stop recording time stamps and to print the results to the UNIX **stderr** device. This printout includes the total time between all two consecutive stamps, as well as the average, maximum and minimum time differences between each two consecutive stamps. This is followed by a list of the differences themselves, printed in whole microseconds, eight per line.

dump_queue_stats prints the values of a number of usage counts maintained by the q-interface. These can be used to determine efficiency in terms of how many actual I/O operations were performed in response to calls to certain q-interface functions.

Appendix

A.1 Sample Programs that Utilize the q-interface

This appendix demonstrates a sample client and a sample server program, both of which utilize the q-interface. The task is a trivial one, designed only to illustrate features of the q-interface. The client repeatedly prompts the user for the name of an input file, opens that file, and sends the name and contents to the server. The server simply prints onto its standard output whatever data is sent to it by the client. The server is capable of handling multiple clients simultaneously, although its printed output will be garbled because it will interleave the data from different clients in the order received.

A.1.1 Common Definitions

This section shows the common header file used by both the client and the server. It defines the "well known port" number for the server, the maximum size of the data messages to be exchanged, and the structure of the message buffer.

```
/*  qcommon.h   - header file for qclient and qserver demos */

#include "queue.h"

#define DEFAULT_SERVER_PORT     ":3456"
#define MAXDATASIZE             256

struct simple_preamble  {
                        void                *reserved;
                        };

struct simple_header    {
                        unsigned int        reserved;
                        unsigned int        data_length;
                        };

struct simple_data      {
                        char                info[MAXDATASIZE];
                        };


#define PREAMBLESIZE    sizeof(struct simple_preamble)
#define HEADERSIZE      sizeof(struct simple_header)
#define MAXMESSAGESIZE  (HEADERSIZE+sizeof(struct simple_data))


struct simple_buffer    {
                        struct simple_preamble      preamble;
                        struct simple_header        header;
                        struct simple_data          data;
                        };
```

A.1.2 Sample Server

This section shows the sample server program. The first thing it does is call **q_setup** to initialize the q-interface. It then calls **q_strtolocation** to construct the location to be used by its listening port from the name given to it as a parameter and from the "well know port" number of the server, passes this

location to **q_openlisteningpost** to establish the listening post, and prints a message to indicate that it has established itself as a server. It then calls **q_signal** to establish the call-back routine **quit_handler** to deal with termination signals, such as control-C from the keyboard, calls **q_acceptconnection** to establish the call-back routine **accept_a_client** to deal with new connections from remote clients, and finally calls **q_loop** to turn control over to the q-interface.

Whenever a client makes a connection, the q-interface invokes the call-back routine **accept_a_client**, which prints the location of the client passed into it in the **buffer** parameter, and then calls **q_readblock** to enqueue a read into a buffer and to establish the call-back routine **serve_a_client** to deal with the data from the new client. Whenever the client sends a message, the q-interface invokes the call-back routine **serve_a_client**, which calls **q_writejustdata** to print the data onto the server's standard out device using the data length field in the message header to know how much data to print. It then calls **q_readblock** to enqueue a read into a new buffer with another call-back to **serve_a_client** when the read completes.

```
/*  qserver.c   - a server program that uses the q-interface */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>

#include "server.h"
#include "qcommon.h"


/*  call-back invoked when user types ^C on terminal */
void quit_handler( int signo, char *buf, unsigned int len )
    {
    printf("server quiting\n");
    q_exit(EXIT_FAILURE);
    }


/*  call-back invoked when client sends us some data */
void serve_a_client( int client_fd, char *buffer,
                                        unsigned int len)
    {
    struct simple_buffer *ptr=(struct simple_buffer *)buffer;


    if( len < HEADERSIZE )
        {/* end of file, close the connection to the client */
        free_a_buffer(buffer);
        printf("server disconnected from client %d\n",
                                        client_fd);
        if( q_close(client_fd) < 0 )
            printf("server unable to close client %d\n",
                                        client_fd);
        }
    else
        {/* actually read some data, print it */
        q_writejustdata(STDOUT_FILENO, buffer,
            ntohl(ptr->header.data_length), release_a_buffer);
```

```
        /*  now set up to read a new buffer from client */
        q_readblock(client_fd, get_a_buffer(), MAXMESSAGESIZE,
                                            serve_a_client);
        }
    }


/*  call-back invoked when client establishes new connection */
void accept_a_client( int client_fd, char *buffer,
                                            unsigned int len )
    {
    printf("server connected to client %d located at %s\n",
            client_fd, q_strlocation((q_location_t *)buffer));

    /*  start reading data from client */
    q_readblock(client_fd, get_a_buffer(), MAXMESSAGESIZE,
                                            serve_a_client);
    }


void server( char *interface_name, int argc, char *argv[] )
    {
    int                 listening_fd;
    q_location_t        address, interface_location;


    q_setup(PREAMBLESIZE, HEADERSIZE, MAXMESSAGESIZE);

    if(q_strtolocation(interface_name, DEFAULT_SERVER_PORT,
                               &interface_location) < 0 )
        {
        if( interface_name != NULL )
            printf("server can't make location from %s\n",
                                       interface_name);
        else
            printf("server can't make default listening post "
                                       location\n");
        q_exit(EXIT_FAILURE);
        }

    /*  establish a listening post for this server */
    if((listening_fd =
          q_openlisteningpost(&interface_location,&address))<0)
        {
        if( interface_name != NULL )
            printf("server can't open listening post from %s: "
                "%s\n",interface_name,strerror(-listening_fd));
        else
            printf("server can't open default listening post: "
                "%s\n", strerror(-listening_fd));
        q_exit(EXIT_FAILURE);
        }
```

```
    /*  have successfully established server listening post */
    printf("server listening at location %s\n",
                                q_strlocation(&address));

    /*  set up to catch SIGINT and SIGQUIT signals */
    if( q_signal(SIGINT, NULL, 0, quit_handler) < 0 )
        q_exit(EXIT_FAILURE);
    if( q_signal(SIGQUIT, NULL, 0, quit_handler) < 0 )
        q_exit(EXIT_FAILURE);

    /*  set up call-back when a client wants to connect */
    q_acceptconnection(listening_fd, get_a_buffer(),
                       MAXMESSAGESIZE, accept_a_client);

    /*  now repeatedly accept client connections or data */
    /*  from now on, control comes to us only via call-backs */
    q_loop();
    }
```

A.1.3 Sample Client

This section shows the sample client program. The first thing it does is call **q_setup** to initialize the q-interface. Next, it uses **q_strtolocation** to create the server location from the name given to it as a parameter (if any) and the default server port number, and calls **q_openclient** to open a connection to that server. Once the connection is established, the client prints a message to that effect, then calls the local routine **prompt** which prompts the user for the name of a file and calls q_readjustdata to enqueue a read to get that input into a buffer. The last thing the client does is to call **q_loop** to turn control over to the q-interface.

When the user types the name of a file, the q-interface will invoke the call-back routine **get_the_name** which was established by **prompt**. This routine uses the name to open the file, then sends the name to the server as a line of text by calling **q_writeblock**. This establishes the call-back routine **do_the_file** which is invoked by the q-interface when the write to the server completes. **do_the_file** simply reads the next block of data from the file and calls **q_writeblock** to send it to the server. When **do_the_file** detects the end of the file, it closes the file and reprompts the user for the next file name.

```
/*  qclient.c   - a client program that uses the q-interface */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <fcntl.h>

#include "client.h"
#include "qcommon.h"


static int  filefd, clientfd;


void get_the_name( int infd, char *name, unsigned int len );
```

```
/*  routine to prompt user, then queue read from keyboard */
void prompt( char *buffer )
    {
    printf("\nenter a file name> ");
    fflush(stdout);
    /*  leave byte at end so we can make input into string */
    q_readjustdata(STDIN_FILENO, buffer, MAXDATASIZE-1,
                                        get_the_name);
    }


/*  call-back invoked after buffer has been sent to server */
void do_the_file( int fd, char *buffer, unsigned int len )
    {
    int     n;
    struct simple_buffer *ptr=(struct simple_buffer *)buffer;

    if( (n = read(filefd, ptr->data.info, MAXDATASIZE)) > 0 )
        {
        ptr->header.data_length = htonl(n);
        q_writeblock(fd, buffer, HEADERSIZE+n, do_the_file);
        }
    else
        {
        close(filefd);
        prompt(buffer);
        }
    }

/*  call-back invoked when user types file name on keyboard */
void get_the_name( int infd, char *buffer, unsigned int len )
    {
    struct simple_buffer *ptr=(struct simple_buffer *)buffer;
    char                    *name;


    if( len == 0 )
        {
        printf("\n");
        q_exit(EXIT_SUCCESS);
        }

    ptr->data.info[len] = '\0';     /* make name into string */
    if( (name = strtok(ptr->data.info, " \t\n\r")) == NULL )
        {
        prompt(buffer);
        }
    else if( (filefd = open(name, O_RDONLY)) < 0 )
        {
        perror(name);
        prompt(buffer);
        }
    else
```

```
        {/* file opened ok, send name to server as text line */
        printf("client sending file %s\n", name);
        len = strlen(ptr->data.info);
        ptr->data.info[len] = '\n';
        len += 1;
        ptr->header.data_length = htonl(len);
        q_writeblock(clientfd, buffer, HEADERSIZE+len,
                                            do_the_file);
        }
    }


void client( char *server_name, int argc, char *argv[] )
    {
    q_location_t        server_address, client_address;


    q_setup(PREAMBLESIZE, HEADERSIZE, MAXMESSAGESIZE);

    if(q_strtolocation(server_name, DEFAULT_SERVER_PORT,
                                  &server_address) < 0 )
        {
        if( server_name != NULL )
            printf("client can't make location from %s\n",
                                  server_name);
        else
            printf("client can't make default server "
                                       "location\n");
        q_exit(EXIT_FAILURE);
        }

    /*  make a connection to the server */
    if((clientfd =
            q_openclient(&server_address,&client_address)) < 0)
        {
        printf("client can't connect to server at %s: %s\n",
           q_strlocation(&server_address),strerror(-clientfd));
        q_exit(EXIT_FAILURE);
        }

    /*  we are now successfully connected to a remote server */
    printf("client at location %s\n",
                           q_strlocation(&client_address));
    printf("client will send to server at location %s\n",
                           q_strlocation(&server_address));

    /*  get file name from user & transmit data to server */
    prompt(get_a_buffer());

    /*  from now on, control comes to us only via call-backs */
    q_loop();
    }
```

A.2 Alphabetic List of q-interface Function Prototypes

The following alphabetic list contains the prototypes of all the q-interface functions, exactly as they appear in the header file "queue.h".

```
extern void dump_buffer_stats( void );

extern char *dup_a_buffer(
        const char *oldbuf,
        unsigned int length );

extern void free_a_buffer( char *ptr );

extern void free_a_header( char *ptr );

extern char *get_a_buffer( void );

extern char *get_a_header( const char *data_ptr );

extern void q_block( void );

extern void q_cancel( int fd,
        int who,
        int do_call_backs );

extern int q_close( int fd );

extern int q_deadlinetimeout( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );

extern void dump_queue_stats( void );

extern void q_endup( void );

extern void q_exit( int status );

extern int q_intervaltimeout( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *interval );

extern int q_listen( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int));

extern int q_locationtostr( q_location_t *location,
        unsigned int namelen,
        char *name );

extern void q_loop( void );
```

```
extern int q_openchildserver( q_location_t *server_request,
        q_location_t *server_addr );

extern int q_openclient( q_location_t *server_addr,
        q_location_t *client_addr );

extern int q_openserver( q_location_t *server_request,
        q_location_t *server_addr );

extern void q_probe( int fd );

extern void q_poll( void );

extern int q_readblock( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int) );

extern int q_readblocktimeout( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );

extern int q_readjustdata( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int) );

extern int q_readjustdatatimeout( int fd,
        char *buffer,
        unsigned int maxlen,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );

extern void q_reset( void );

extern void q_setup( unsigned int pre_size,
        unsigned int min_size,
        unsigned int max_size );

extern int q_signal( int signo,
        char *buffer,
        unsigned int len,
        void (*call_back)(int, char *, unsigned int) );

extern void q_start_stamp( int fd,
        int mode,
        unsigned int max_stamp );

extern void q_stop_stamp( unsigned int max_stamp_print );
```

```
extern char *q_strlocation( q_location_t *location );

extern int q_strtolocation( const char *name,
        const char *default_name,
        q_location_t *location );

extern int q_writeblock( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int) );

extern int q_writeblocktimeout( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );

extern int q_writejustdata( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int) );

extern int q_writejustdatatimeout( int fd,
        char *buffer,
        unsigned int length,
        void (*call_back)(int, char *, unsigned int),
        struct timeval *deadline );

extern void q_wait( struct timeval *timeout );
```

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399