

AMarquee

Jeremy Friesner

COLLABORATORS

	<i>TITLE :</i> AMarquee		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Jeremy Friesner	August 5, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	AMarquee	1
1.1	Contents	1
1.2	miami	1
1.3	residentnote	2
1.4	Installation	3
1.5	amitcpinstall	3
1.6	credits	4
1.7	How to reach me	4
1.8	disclaimer	4
1.9	AMarquee is DonationWare	5
1.10	What you need	6
1.11	introduction	6
1.12	Using AMarqueed	7
1.13	Using amarquee.library	9
1.14	api	9
1.15	usingintro	10
1.16	qmessages	11
1.17	qnewsession	14
1.18	qnewsessionasync	15
1.19	qnewhostsession	17
1.20	qnewserversession	19
1.21	qfreesession	20
1.22	qdebugop	21
1.23	qgetop	22
1.24	qdeleteop	23
1.25	qrenameop	24
1.26	qsubscribeop	25
1.27	qsetop	26
1.28	qmessageop	28
1.29	qstreamop	29

1.30	qclearsubscriptionop	31
1.31	qpingop	31
1.32	qinfoop	32
1.33	qsetaccessop	34
1.34	qsetmessageaccessop	35
1.35	freeqmessage	36
1.36	qgo	37
1.37	installdaemon	38
1.38	exampleclients	39
1.39	Thanks	40
1.40	faq	41
1.41	The ground was littered with squashed bugs...	41
1.42	What's Next?	43
1.43	unnamed.1	43
1.44	Known Bugs and Other Problems	43
1.45	otherprogs	44

Chapter 1

AMarquee

1.1 Contents

AMarquee V1.43 by **Jeremy Friesner**

AMarquee is a mechanism by which Amiga programs on multiple computers may easily communicate data with each other. It consists of two parts, an AmiTCP daemon and a shared library. AMarquee has been made as general as possible, and it is my hope that a variety of uses will be found for it.

Disclaimer Don't blame me!

Distribution AMarquee is DonationWare!

Requirements What do I need to run this program?

Introduction What does AMarquee do?

Installation How do I set AMarquee up?

Using AMarquee How to use the amarquee.library

Using AMarqueed How to use the AMarqueed server

Example clients Silly little applets to play with

Credits Where it's due

Acknowledgments Thanks to...

History Bug fixes and enhancements

Future What next?

F.A.Q. Frequently Asked Questions

Known Problems Bugs! Aack!

Other programs Plug, plug!

1.2 miami

If you wish to install AMarquee manually for use with Miami, the procedure is slightly different.

The client side of the AMarquee system, `amarquee.library`, installs exactly the same as with AmiTCP--just copy it into LIBS:.

To install the server, you must do something different. With AmiTCP, you have two config files, `amitcp:db/services`, and `amitcp:db/inetd.conf`. To install a new daemon, you edit these files with your text editor, as described in the Installation section. In Miami, however, these files are actually windows in the GUI. What you need to do is add the same lines to the ListViews in these windows that would have been added to the corresponding config files under AmiTCP.

To wit:

- 1) Open up the "Miami" application in your Miami drawer.
 - 2) Click on "Database" on the ListView at the left. The rest of the window changes to display a new interface.
 - 3) Make sure the cycle gadget at the top of the window is set to "services"
 - 4) Click the "Add" button near the lower right, and type the line `AMarquee 2957/tcp` into the string gadget, and press return. Verify that this line is now listed at the bottom of the ListView.
 - 5) Select "InetD" in the cycle gadget at the top of the window. The ListView will now show a different list of entries.
 - 6) Click the "Add" button near the lower right, and type the line `AMarquee stream tcp nowait root amitcp:serv/AMarqueed` into the string gadget, and press return. The line you just typed should now be included at the bottom of the ListView's list.
- [Read this note about making AMarqueed resident](#)
- 7) Select "Save" from the "Settings" menu to make your additions permanent.

1.3 residentnote

There are two ways to have the `AMarqueed` executable be run when an AMarquee client connects to your computer: resident, and non-resident. If the `AMarqueed` executable is not resident, a new copy of its binary code will be loaded into memory for each connection that is accepted. If, on the other hand, it has been made resident, then only one copy of the `AMarqueed`

executable will ever be loaded into memory, no matter how many simultaneous connections there are. Obviously, making AMarqueued resident will save memory in most situations.

There are two things you have to do to make AMarqueued run resident. The first is to make sure the line
resident amitcp:serv/AMarqueued
has been executed before your first AMarquee connection is accepted. This can be done in your user-startup file, or in your amitcp:bin/startnet file, or anywhere else, as long as it is done somewhere. (The included Installer script puts this command in your user-startup file)

The second thing to do is modify AMarquee's line in the InetD configuration so that no file path is specified. This is necessary because specifying a file path will cause the executable to be loaded from that path rather than re-used from memory. Thus, you should edit the file amitcp:db/inet.conf (or the Databases/InetD window in Miami) to contain this line:
AMarquee stream tcp nowait root AMarqueued
(note that the amitcp:serv/ prefix in the last field has been deleted!)

You can check to see if the residenting is working by accepting several AMarquee connections, and then typing "resident" at a shell prompt. You should see AMarqueued listed, with a usage count of greater than 0.

1.4 Installation

If you are using AmiTCP or Miami, you can use the included Installer script to install AMarquee.

If you don't like to use Installer scripts, there are manual installation instructions available for [AmiTCP](#) and [Miami](#) .

1.5 amitcpinstall

AMarquee is made of two parts, each of which may be installed independantly of the other. The first part, amarquee.library, is a front-end library for use by Amiga programs that wish to act as AMarquee clients. It may be installed by copying it to your LIBS: directory.

The second part, AMarqueed, is the TCP server program that stores broadcasted information for other Amigas and co-ordinates information updating and notification. You only need to install it if you wish to use your Amiga as an AMarquee server.

To install the AMarquee server for use with AmiTCP, do the following:

- 1) Copy the file AMarqueed from the distribution into your amitcp:serv directory.
- 2) Add the following line to the end of your amitcp:db/services file:
AMarquee 2957/tcp
- 3) Add the following line to the end of your amitcp:db/inetd.conf file:
AMarquee stream tcp nowait root amitcp:serv/AMarqueed
Read this note about making AMarqueed resident
- 4) Re-start AmiTCP. Your AMarquee server should now be ready to accept connections. Try running one of the included **example** programs to "localhost" to test it out.

1.6 credits

AMarquee V1.43

Created by **Jeremy Friesner**

Compiled with DICE C by Matt Dillon

Uses AmiTCP, by Network Solutions Development, Inc.

1.7 How to reach me

Here are some ways to get in touch with me:

by EMail: jfriesne@ucsd.edu

jaf@chem.ucsd.edu

by SMail: Jeremy Friesner

4680 Mt. Longs Drive

San Diego, CA 92117

1.8 disclaimer

This software comes with no warranty, either expressed or implied.

The **author** is in no way responsible for any damage or loss that may occur due to direct or indirect usage of this software. Use this software entirely at your own risk.

1.9 AMarquee is DonationWare

NOTE: Please **report** any bugs you find while using this software.

AMarquee may be distributed freely, as long as the original archive is kept intact.

AMarquee is DonationWare. I've put a lot of time into it to make it as easy-to-use, stable, efficient, and useful as possible, so if you find AMarquee to your liking and use it often, please consider sending **me** a \$5 or \$10 donation, and in return I will send you the source code to AMarquee (if you want it), future upgrades directly and give your suggestions preferred treatment. Also, if you write a program that benefits from AMarquee and nets you a significant amount of money, I request that you think about kicking a small percentage of the profits down to me!

However, if you can't afford that or for some other reason don't want to send money, that's okay also. Just send me email telling me that you're using it, and list any suggestions that you have for improving it. :-)

Permission is given to include this program in a public archive (such as a BBS, FTP site, PD library or CD-ROM) providing that all parts of the original distribution are kept intact. These are as follows:

Listing of archive 'AMarquee1.43.lha':

Original Packed Ratio Date Time Name

```
-----
1233 595 51.7% 27-Apr-97 23:54:10 AMarquee.info
89164 27220 69.4% 27-Apr-97 23:54:10 +amarquee.guide
1542 1096 28.9% 27-Apr-97 23:54:10 +AMarquee.guide.info
16612 9014 45.7% 27-Apr-97 23:54:10 +amarquee.library
4194 1926 54.0% 27-Apr-97 23:54:08 +AMarquee.readme
835 268 67.9% 27-Apr-97 23:54:10 +AMarquee.readme.info
34132 18486 45.8% 27-Apr-97 23:54:10 +AMarqueed
4123 1604 61.0% 27-Apr-97 23:54:12 +EditTextFile.rexx
10176 5932 41.7% 27-Apr-97 23:54:12 +AMarqueeDebug
6016 2041 66.0% 27-Apr-97 23:54:12 +amarqueedebug.c
9200 5494 40.2% 27-Apr-97 23:54:14 +AMarqueeHost
4175 1563 62.5% 27-Apr-97 23:54:14 +amarqueehost.c
11868 6922 41.6% 27-Apr-97 23:54:14 +AMarqueeServer
5307 1924 63.7% 27-Apr-97 23:54:12 +AMarqueeServer.c
7400 4731 36.0% 27-Apr-97 23:54:12 +BounceCount
2910 1167 59.8% 27-Apr-97 23:54:14 +BounceCount.c
247 172 30.3% 27-Apr-97 23:54:14 +dmakefile
```

```

10408 6410 38.4% 27-Apr-97 23:54:12 +MiniIRC
6379 2171 65.9% 27-Apr-97 23:54:16 +MiniIRC.c
6984 4444 36.3% 27-Apr-97 23:54:12 +RemoveTest
2546 1035 59.3% 27-Apr-97 23:54:14 +RemoveTest.c
9324 5734 38.5% 27-Apr-97 23:54:14 +SillyGame
8214 2586 68.5% 27-Apr-97 23:54:14 +SillyGame.c
7404 4711 36.3% 27-Apr-97 23:54:14 +StreamCheck
3132 1242 60.3% 27-Apr-97 23:54:14 +StreamCheck.c
7216 4569 36.6% 27-Apr-97 23:54:14 +StreamGen
2422 1018 57.9% 27-Apr-97 23:54:14 +streamgen.c
7456 4717 36.7% 27-Apr-97 23:54:12 +SyncTest
3250 1302 59.9% 27-Apr-97 23:54:16 +SyncTest.c
2162 579 73.2% 27-Apr-97 23:54:16 +AMarquee_protos.h
808 307 62.0% 27-Apr-97 23:54:16 +amarquee.fd
2470 1091 55.8% 27-Apr-97 23:54:16 +AMarquee.h
1166 372 68.0% 27-Apr-97 23:54:16 +AMarquee_pragmas.h
12811 4030 68.5% 27-Apr-97 23:54:10 +Install_AMarquee
612 329 46.2% 27-Apr-97 23:54:12 +Install_AMarquee.info

```

303898 136802 54.9% 27-Apr-97 23:54:44 35 files

No charge may be made for this program, other than a reasonable copying fee, and/or the price of the media.

1.10 What you need

- Kickstart V37 (WorkBench 2.04) or higher,
- AmiTCP3.0b2 or higher (or a compatible TCP stack)
- A C compiler, if you wish to write AMarquee programs.

1.11 introduction

Please read the [History](#) section for information on changes and bug-fixes.

AMarquee is a system for broadcasting information between Amigas. It uses a server-hub information storage model, where any Amiga program may upload information to the central server, and other Amiga programs may download it. Information is stored on the server in a filesystem-like structure, with each client given its own "directory" in the tree. Each client may read or write to its own directory on the server, and may read

the directories of the other clients (assuming they allow it access). The AMarquee system was designed in such a way that "polling" of information to detect updates should never be necessary. Instead, each client may **subscribe** to a set of entries in the server, and whenever the subscribed data is updated, the client will be notified of the change. This technique helps keep bandwidth usage as low as possible.

As alternatives to the information-storage model, AMarquee features a message-passing model and a direct-connect model. You may now easily pass data **messages** to other clients, through the AMarquee server, or **connect directly** to other AMarquee clients and send data straight to them, without bothering the server.

Also, the AMarquee system is heavily multithreaded. A client-side thread is started in the background for each connection made with `amarquee.library`, and a new server thread is created on the server for each connection received. This multithreading allows the user program to do other things while data is being sent or received, and avoids bottlenecks on the server computer. The multithreading is completely transparent to the user program. Furthermore, with `amarquee.library`, no socket programming knowledge is necessary. All the user's code needs to do is make `amarquee.library` calls and `Wait()` on a supplied Exec-style `MsgPort` for data.

Lastly, both the AMarquee client code and the Amarqueed server code are re-entrant, to minimize memory usage.

1.12 Using Amarqueed

Amarqueed is the "server" portion of the AMarquee system. Its job is to act as the client program's "proxy" or representative on the server computer. It stores data that the client uploads, and returns data that the client has requested, either directly or in response to "subscribed" data having been changed.

Once you have AMarqueed **installed**, it should mostly take care of itself. However, you can (and probably should) specify some parameters for AMarqueed to use. AMarqueed looks for the following ENV variables on startup:

- `AMARQUEED_MAXMEM`

If set, the value of this variable will be taken as the maximum number of kilobytes each daemon is allowed to allocate. For example, typing

"setenv AMARQUEED_MAXMEM 45" limits each connection to allocating no more than 45K of memory for data storage.

- AMARQUEED_MINFREE

If set, the value of this variable will be taken as the size of a "safety buffer" of free memory. No AMarqueued process will be able to allocate more memory unless at least this much free memory exists in the system. For example, entering "setenv AMARQUEED_MINFREE 100" ensures that AMarqueued processes will never use up the last 100K of system memory.

- AMARQUEED_MAXCONN

If set, this variable determines the maximum number of simultaneous connections that will be allowed from any given host. This can be used to prevent any one computer from "hogging" the server's capacity.

- AMARQUEED_TOTALMAXCONN

If set, this variable determines the maximum number of simultaneous AMarqueued connections allowed. For example, entering "setenv AMARQUEED_TOTALMAXCONN 5" will ensure that there are never more than 5 AMarqueued processes running at once.

- AMARQUEED_BANNED

If there are some clients you do not wish to allow to use your server, you can specify that they not be banned with this option. The environment variable should be set to a regular expression in the form of /foo/bar. Thus, if you wanted to deny access to all programs named "Bob" originating from commercial sites, you could do a setenv AMARQUEED_BANNED /#?.com/Bob

Or, to ban all AMarquee connections from evil.hackers.com, do setenv AMARQUEED_BANNED /evil.hackers.com/#?

Or, if you wanted to dedicate your server solely to the CoolApp client:

```
setenv AMARQUEED_BANNED /#?/~-(CoolApp)
```

Note that the value of this variable must start with an initial '/'!

- AMARQUEED_PRIORITY

If you wish the AMarqueued server tasks to run at a particular priority, you can set this variable to the priority you want them to run at. If this is not set, AMarqueued daemons will run at the priority AmiTCP launches them with (-10 on my system).

- AMARQUEED_PINGRATE

In order to keep the shared data tree free of clutter, AMarqueued makes sure each client is still there by sending it an empty transaction every so often. While these transactions are transparent

to user programs, they do have a slight impact on network and CPU usage. This ENV variable allows you to set the number of minutes of idle time that will elapse before a null transaction is sent. For example, setting AMARQUEED_PINGRATE to 5 will cause a null transaction be sent to each client after 5 minutes of idle time. Then, if the client has not responded to the transaction within 5 more minutes, it will be removed from the system. The default rate is every 3 minutes.

- AMARQUEED_DEBUG

If set, each AMarqueued session will open a debug console on startup, showing various state information. [Note that you must do something like "setenv AMARQUEED_DEBUG 1" for this to take effect, just entering "setenv AMARQUEED_DEBUG" won't do it.]

- AMARQUEED_FAKECLIENT

This option may be set to a host/program name path (e.g. "/fakehost/fakeprogram", in which case the AMarquee server will attempt to give incoming connections this designation instead of their actual one. This feature allows you to easily simulate connections from various hosts when debugging an AMarquee program. It should not be set during normal use.

1.13 Using amarquee.library

Note: amarquee.library was written and compiled using DICE C.

While the library itself should be compatible with programs written using any standard Amiga compiler, you may have to slightly modify the included header files to match your compiler's tastes.

[Introduction to amarquee.library](#)

[How to interpret QMessages](#)

[The amarquee.library API](#)

1.14 api

struct QSession * [QNewSession](#) (char * hostname, LONG port, char * progname)

struct QSession * [QNewSessionAsync](#) (char * hostname, LONG port, char * progname)

struct QSession * [QNewHostSession](#) (char * hostnames, LONG * port, char * prognames)

struct QSession * [QNewServerSession](#) (char * hostnames, LONG port, char * prognames)

void **QFreeSession** (struct QSession * session)
LONG **QDebugOp** (struct QSession * session, char * string)
LONG **QGetOp** (struct QSession * session, char * wildpath, LONG maxBytes)
LONG **QDeleteOp** (struct QSession * session, char * wildpath)
LONG **QRenameOp** (struct QSession * session, char * path, char * label)
LONG **QSubscribeOp** (struct QSession * session, char * wildpath, LONG maxBytes)
LONG **QSetOp** (struct QSession * session, char * path, void * buffer, ULONG bufferLength)
LONG **QStreamOp** (struct QSession * session, char * path, void * buffer, ULONG bufferLength)
LONG **QClearSubscriptionsOp** (struct QSession * session, LONG which)
LONG **QPingOp** (struct QSession * session)
LONG **QInfoOp** (struct QSession * session)
LONG **QSetAccessOp** (struct QSession * session, char * newAccess)
LONG **QSetMessageAccessOp** (struct QSession * session, char * newAccess, LONG maxBytes)
LONG **QMessageOp** (struct QSession * session, char * hosts, UBYTE * buffer, ULONG bufferLength)
LONG **QGo** (struct QSession * session, ULONG flags)
void **FreeQMessage** (struct QSession * session, struct QMessage * qmsg)

1.15 usingintro

The AMarquee library allows you to do network broadcasting in an asynchronous, multithreaded manner, with almost no knowledge of TCP programming. It does this by creating a background process that handles the TCP transmission and reception for you, and sends you messages (called QMessages) to notify you of data you have recieved. Each of the Q*Op() functions in AMarquee.library creates a message with your data in it, and sends it to the TCP thread for queuing and eventual transmission. With this setup, your code need never wait while data is transmitted or recieved, unless it wants to.

It is important to understand how data is accessed in the AMarquee system. All AMarquee public data is stored in a tree on the server, and each node in the tree has a name, which is a null-terminated string. Each node may be referenced via a "regular node path string", which is somewhat like a UNIX file path. Each client program that connects to the AMarquee server is automatically given its own "directory" node in the tree, based on the IP name of the computer it is running on, and the name it has chosen for itself. Thus, if you are running an AMarquee-based client program on a computer named mycomputer.mycompany.com, and it connects to the server as "MyProgram", then the program's

"home directory" would be

```
/mycomputer.mycompany.com/MyProgram
```

And if you were to add a node named "MyData" to your "directory", it would show up as

```
/mycomputer.mycompany.com/MyProgram/MyData
```

All nodes in the server's data tree may be accessed or referred to by strings such as these. Also, each node in the data tree contains a data buffer of variable size. (So much for the filesystem analogy-- nodes in this tree can be both "files" and "directories" at once!)

This data buffer is a simple, raw array of bytes, and hence may contain any data you wish to keep in it. The data buffer of the "home directory" node is hard-coded to a null-terminated string containing the IP number of your host computer, but the data buffers of all other nodes may be set arbitrarily.

Also: There is a "short" method of specifying nodes that are located within your own directory space. If you specify a path string without an initial slash, it will be assumed you mean the path is relative to your home node. Thus, specifying "MyData/DataItem1" as a node path would be the same (for most purposes) as specifying

```
/mycomputer.mycompany.com/MyProgram/MyData/DataItem1
```

Lastly, AMarquee makes heavy use of the Amiga wildcarding system to specify groups of nodes. Thus, to specify all client programs running on all connected Amigas, you could use

```
/#?/#?
```

Or to specify all entries under the node "MyData", that begin with the string "DataItem", from computers in Australia, that are connected via an AMarquee client registered as MyProgram, you could do:

```
/#?.au/MyProgram/MyData/DataItem#?
```

All matching and storage of nodes is case sensitive!!

Probably the best way to get a feel for how AMarquee works is to look at and play around with the example programs in the [examples](#) directory of this distribution.

1.16 qmessages

All communications from the client TCP thread and the AMarquee server arrive at your application's doorstep in the form of QMessages.

These QMessages can be accessed by your program by waiting on

the `qMsgPort` field in the `QSession` struct that is returned by `QNewSession`. Every `QMessage` you receive should eventually be freed by your application, via the `FreeQMessage` call.

You must use this call, and not `FreeMem()` or `ReplyMsg()`, when you are done with `QMessages`. Any data you wish to keep from the `QMessage` you must have been copied out into your own storage area(s) before you free the `QMessage`.

The `QMessage` has many fields, all of which bear explanation.

Here they are:

- struct `Message` `qm_Msg`;

Used for normal Amiga messaging stuff. You shouldn't need to access the data within this struct directly.

- LONG `qm_ID`;

Contains the ID number of a transaction you sent, that this message is in response to. ID numbers for each transaction are returned by the `Q*Op()` calls, and let you determine which `QMessages` resulted from which calls. Valid ID numbers are monotonically increasing, starting at 1. You may see `qm_ID`'s of 0, if the message is not associated with any particular transaction (such as when the message was caused by a `QMessageOp` executed by another client), or of -1 for certain types of error.

- int `qm_Status`;

Contains the error status of the message. For normal operation results, this will be set to `QERROR_NO_ERROR`, but if there was an error, it may be one of the following:

`QERROR_UNKNOWN`

Something has gone wrong, but nobody knows what! You should actually never see this.

`QERROR_MALFORMED_KEY`

You specified a node path that was syntactically incorrect, or is impossible (such as specifying a node in a directory that doesn't exist)

`QERROR_NO_SERVER_MEM`

The AMarquee server ran out of memory and thus was not able to perform the operation. This is a serious error, and if you get it you should not count on any subsequent transactions in the same message packet having been performed!

`QERROR_NO_CONNECTION`

The TCP connection to the AMarquee server has been severed. Once

you get this, the only thing you can really do is `QFreeSession` your session and start a new one--there is no way to reconnect a disconnected session.

QERROR_UNPRIVILEGED

You tried to do something you're not allowed to do (such as writing to another client's directory or renaming your root node).

Note that trying to read from the directory of a client who has excluded you via `QSetAccessOp` will not result in this error--rather, you just won't get any results back. You will also get this error code if you sent a `QMessageOp` that no other clients were willing to receive.

QERROR_UNIMPLEMENTED

You tried to use a feature that I haven't implemented yet.

You may get this error if you are receiving data in a direct client-to-client connection and the sending client sends you a transaction that does not make sense in this context.

(for example, a `QSetAccessOp`)

QERROR_NO_CLIENT_MEM

Your computer ran out of memory, and thus could not perform the operation.

- char * qm_Path;

If the `QMessage` is returning information about a particular node, this will point to a NUL-terminated, fully qualified path string identifying the node. As returned by the AMarquee server, this string will never have wildcards in it, and will always be "absolute" (with the computer and program name explicitly specified). If no node is being referenced (as in, for example, error or ping messages), this entry will be NULL.

- void * qm_Data;

If a data buffer is being returned with the `QMessage`, this will point to the first byte of the buffer (otherwise it will be NULL).

The data buffer belongs to the TCP thread, and will be freed when the `QMessage` is freed. You are allowed to read from or write to this buffer until you call `FreeQMessage` on this `QMessage`.

- ULONG qm_DataLen;

Contains the number of bytes in the `qm_Data` buffer that is included with this `QMessage`.

- ULONG qm_ActualLen;

Contains the size of the data buffer associated with this node,

as it is stored on the server. This value may be larger than `qm_DataLen` if you told `QGetOp` or `QSubscribeOp` to limit the size of data buffers you receive (via the `maxBytes` argument).

- `ULONG qm_ErrorLine;`

If `qm_Status` is not `QERROR_NO_ERROR`, this entry will contain the line number of the command in the AMarqueued source code that triggered the error. This is to help me in debugging the AMarquee server.

- `void * qm_Private;`

Points to secret magic stuff used by the TCP thread. Leave this alone!

1.17 qnewsession

`amarquee.library/QNewSession` `amarquee.library/QNewSession`

NAME

`QNewSession` - Creates a new connection to an AMarqueued server.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
struct QSession * QNewSession(char * hostname, LONG port, char * progname)
```

FUNCTION

Creates a new TCP-handling thread and attempts to connect it to the specified AMarqueued server.

NOTE

This function is synchronous--it will not return until either a TCP connection has been made to the remote host, or the attempt has failed. If the connection fails, this function will return `NULL` with no side effects.

If a non-`NULL` `QSession` is returned, it must be `QFreeSession`'d before you close the `amarquee.library`.

INPUTS

`hostname` - The IP name of the computer to connect to. (e.g. `foo.bar.com`)

`port` - The port to connect on. Port 2957 is the "official" AMarqueued port.

`progname` - A null-terminated string indicating the name the client program wishes to use. If another client from your host has already registered `progname`, the server for that client will close its connection and quit so that your client program

will still be uniquely addressable.

RESULTS

On success, returns a pointer to a `QSession` struct that should be passed to the other functions in this API. The `QSession` struct also contains a pointer to an `exec.library MsgPort` that your program should `Wait()` on and `GetMsg()` from to receive `QMessages` from the TCP thread. Returns `NULL` if a connection could not be established.

EXAMPLE

```
struct QSession * s;
if (s = QNewSession("isys.ml.org", 2957, "ExampleProgram"))
printf("Connection to isys.ml.org:2957 was successful\n");
else
printf("Connection failed.\n");
```

SEE ALSO

[QNewSessionAsync](#) , [QNewHostSession](#) , [QNewServerSession](#) , [QFreeSession](#)

1.18 qnewsessionasync

amarquee.library/QNewSessionAsync amarquee.library/QNewSessionAsync

NAME

`QNewSessionAsync` - Starts a TCP connection thread, returns immediately.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
struct QSession * QNewSessionAsync(char * hostname, LONG port, char * progname)
```

FUNCTION

This function works the same as `QNewSession` , except that instead of waiting for the TCP connection to be fully established, it returns immediately.

When (and if) the TCP connection does connect to the target computer, your program will be notified via a `QMessage` . This `QMessage` will have a `qm_ID` of zero, contain your program's "home directory" as the `qm_Path` field (e.g. `"/yourhostname/yourprogname"`), and have `""` as the `qm_Data` field.

If the TCP connection fails, you will be sent a `QMessage` with the `qm_Status` field set to `QERROR_NO_CONNECTION`.

NOTE

If a non-`NULL` `QSession` is returned, it must be `QFreeSession` 'd before you close the `amarquee.library`.

You may call `QFreeSession()` at any time, whether the returned `QSession` has connected or not.

Any transactions (`Q*Op()`, `QGo()`, etc) that you send to a `QSession` that is still connecting will not be processed until after the TCP connection has been established.

INPUTS

`hostname` - The IP name of the computer to connect to. (e.g. `foo.bar.com`)

`port` - The port to connect on. Port 2957 is the "official" AMarquee port.

`programe` - A null-terminated string indicating the name the client program wishes to use. If another client from your host has already registered `programe`, the server for that client will close its connection and quit so that your client program will still be uniquely addressable.

RESULTS

On success, returns a pointer to a `QSession` struct that should be passed to the other functions in this API. The `QSession` struct also contains a pointer to an `exec.library MsgPort` that your program should `Wait()` on and `GetMsg()` from to receive **QMessages** from the TCP thread. Returns `NULL` if a connection thread could not be established.

EXAMPLE

```
struct QSession * s;
/* demonstrates how you can Wait() on both the pending connection,
and other events (here, a CTRL-C) at the same time */
if (s = QNewSessionAsync("isys.ml.org", 2957, "ExampleProgram"))
{
printf("Connecting to isys.ml.org:2957 ....\n");
while(1)
{
struct QMessage * qMsg;
ULONG signals = (1L << s->qMsgPort->mp_SigBit) | (SIGBREAKF_CTRL_C);
/* Wait for next message from the server */
signals = Wait(signals);
if (signals & (1L << s->qMsgPort->mp_SigBit))
{
while(qMsg = (struct QMessage *) GetMsg(s->qMsgPort))
{
```

```

if (qMsg->qm_Status == QERROR_NO_ERROR) printf("Connection established!\n");
if (qMsg->qm_Status == QERROR_NO_CONNECTION) printf("Connection failed.\n");
}
}
if (signals & SIGBREAKF_CTRL_C)
{
printf("User aborted.\n");
break;
}
}
QFreeSession(s);
}
else
printf("Connection thread failed.\n");

```

SEE ALSO

[QNewSession](#) , [QNewHostSession](#) , [QNewServerSession](#) , [QFreeSession](#)

1.19 qnewhostsession

amarquee.library/QNewHostSession amarquee.library/QNewHostSession

NAME

QNewHostSession - Start a TCP thread waiting for connections

on a port.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
struct QSession * QNewHostSession(char * hostnames, LONG * port, char * prognames)
```

FUNCTION

This function allows your AMarquee client to accept direct TCP connections from other AMarquee clients. It will start a new execution thread, which will listen to the port you specify until an AMarquee thread attempts to connect to it. When a thread does connect to its port, it will verify that the connecting thread matches the criteria specified via the "hostnames" and "prognames" arguments, and if the connecting program passes, it will send you a [QMessage](#) to tell you that a connection has been received. This [QMessage](#) will have a qm_ID of zero, and the qm_Path field will contain the path of the connecting client's root node. (e.g. "/computername/progname"). Also, the qm_Data field will contain the program's IP address,

as a string. After this **QMessage** has been received, you may send data messages to the connected client via **QSetOp** , **QDeleteOp** , **QPingOp** , **QStreamOp** , or **QInfoOp** , and receive **QMessages** in the normal way. Any other operations will cause the receiving client to get a **QERROR_UNIMPLEMENTED** error message.

NOTE

This function will return as soon as the listening port has been set up. Until someone connects to its port, however, the **QSession** returned is "dormant", and any transactions sent to it will result in **QERROR_NO_CONNECTION** **QMessages**.

(Note that you can still **QFreeSession** the dormant **QSession**, if you get tired of waiting for a connection)

Once a connected **QSession** has been disconnected, it can no longer be used for anything and should be freed. If you wish to receive another connection, you will need to call **QNewHostSession** again.

Don't forget to call **QFreeSession** 'd on all allocated **QSessions** before you close **amarquee.library**!

This function requires v38+ of **amarquee.library**.

INPUTS

hostnames - A regular expression determining which hosts are allowed to connect to this **QSession**. For example, "#?" would allow any host to connect, or "#?.edu" would allow only hosts from schools to connect.

port - Pointer to a **LONG** that contains the port to listen on.

Clients who wish to connect to your host session must specify this port number when they call **QNewSession** . If the **LONG** contains the value 0L, then the TCP stack will choose an available port for you and write it into your variable before this function returns.

prognames - A regular expression determining which program names are allowed to connect to this **QSession**. The connecting program's name is specified by the connecting program when it calls **QNewSession** .

RESULTS

On success, returns a pointer to a **QSession** struct that may be used with the other functions in this API. The **QSession** struct also contains a pointer to an **exec.library** **MsgPort** that your program should **Wait()** on and **GetMsg()** from to receive **QMessages** from the TCP thread. Returns **NULL** if a listening port could not be established

(likely because another program was using the port you wanted!).

EXAMPLE

```
struct QSession * s;
LONG port = 0; /* Let the system choose a port for me */
if (s = QNewHostSession("#?", &port, "#?"))
printf("Now listening for connections on port %li.\n",port);
else
printf("Could not open a new host session.\n");
```

SEE ALSO

[QNewSession](#) , [QNewSessionAsync](#) , [QNewServerSession](#) , [QFreeSession](#)

1.20 qnewserversession

amarquee.library/QNewServerSession amarquee.library/QNewServerSession

NAME

QNewServerSession - Link to a socket provided to your program

by inetd when you have been launched as

a server daemon by AmiTCP.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
struct QSession * QNewServerSession(char * hostnames, char * prognames)
```

FUNCTION

This function allows you to create AmiTCP/inetd style daemon programs that use amarquee.library to communicate with other amarquee programs. This function should be called only once, per execution, near the beginning of your program. It will return a QSession that you can use to communicate with the amarquee client that connected to your computer. Your app will receive an initial message containing the identity of the connecting program (e.g. "/computer.name/progName") in the qm_Path field. After that communication takes place in the same way used by connections made via [QNewHostSession](#) .

NOTE

This function will return NULL if your program was not started by inetd, or if the connecting program does not meet the criteria specified in the arguments to this function.

Once a connected QSession has been disconnected, it can no longer be used for anything and should be freed. You cannot receive a second connection with this function (a second connection will

launch a second instance of your program instead).

Don't forget to call `QFreeSession` on all allocated `QSessions` before you close `amarquee.library`!

This function requires v38+ of `amarquee.library`.

For an example of how to use this function, see the file

`AMarqueeServer.c` in the [examples directory](#) .

INPUTS

`hostnames` - A regular expression determining which hosts are allowed to connect to this `QSession`. For example,

"#?" would allow any host to connect, or "#?.edu"

would allow only hosts from schools to connect.

`prognames` - A regular expression determining which program names are allowed to connect to this `QSession`. The connecting program's name is specified by the connecting program

when it calls `QNewSession` .

RESULTS

On success, returns a pointer to a `QSession` struct that may be used with the other functions in this API. The `QSession` struct also contains a pointer to an `exec.library` `MsgPort` that your program should `Wait()` on and `GetMsg()` from to receive `QMessages` from the TCP thread. Returns `NULL` if a listening port could not be established (likely because another program was using the port you wanted!).

EXAMPLE

```
struct QSession * s;
if (s = QNewServerSession("#?", "#?"))
printf("Got the session from InetD. Ready to go!\n");
else
printf("Couldn't get session from InetD.\n");
```

SEE ALSO

[QNewSession](#) , [QNewSessionAsync](#) , [QNewHostSession](#) , [QFreeSession](#)

1.21 qfreesession

`amarquee.library/QFreeSession` `amarquee.library/QFreeSession`

NAME

`QFreeSession` - Closes the given connection to an `AMarquee` server.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
void QFreeSession(struct QSession * session)
```

FUNCTION

Closes the given AMarquee connection, terminates the associated TCP handling thread, and cleans up.

NOTE

You MUST `QFreeSession` every session that you created with any of the `QNew*Session()` functions, before closing `amarquee.library!` Otherwise, you will quickly receive a visit from the Guru, as the TCP thread associated with the `QSession` tries to execute code that is no longer loaded into memory...

Also, all `QMessages` that you have `GetMsg`'d from the `qMsgPort` of "session" must have been `FreeQMessage` 'd before you call this function. Queued transactions and non-`GetMsg()`'d `QMessages` will be automatically freed, however.

INPUTS

`session` - Pointer to the `QSession` struct you wish to free.

RESULTS

None.

EXAMPLE

```
struct QSession * s;
if (s = QNewSession("isys.ml.org", 2957, "ExampleProgram"))
{
/* ... do stuff with the session here ... */
/* ... */
QFreeSession(s);
}
```

SEE ALSO

[QNewSession](#) , [QNewSessionAsync](#) , [QNewHostSession](#) , [QNewServerSession](#)

1.22 qdebugop

`amarquee.library/QDebugOp` `amarquee.library/QDebugOp`

NAME

`QDebugOp` - Sends a debug string to the AMarquee server.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QDebugOp(struct QSession * session, char * string)
```

FUNCTION

Causes the AMarquee server to display the given string in its debug output window, if the debug output window is

open.

NOTE

Probably not all that useful!

INPUTS

session - The session to send the debug op to.

string - The debug string to send to the server.

RESULTS

Returns the assigned ID number for the debug operation on success, or 0 on failure.

EXAMPLE

LONG transID;

```
if (transID = QDebugOp(session, "Just a debug string"))
printf("Debug op succeeded, was given id #%%li\n",transID);
else
printf("Debug op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#)

1.23 qgetop

amarquee.library/QGetOp amarquee.library/QGetOp

NAME

QGetOp - Retrieve a set of entries from the AMarquee server.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QGetOp(struct QSession * session, char * wildpath, LONG maxBytes)
```

FUNCTION

This function instructs the AMarquee server to retrieve the data associated with the given wildpath and download it.

Once the results are downloaded, they will be sent to your process asynchronously as [QMessages](#) .

NOTE

It is sometimes difficult to tell when your QGetOp query is complete and you have received all your results from the query. Since each op is processed in order, you can do this by sending a [QPingOp](#) after the QGetOp, or by specifying TRUE for the sendSync argument in the [QGo](#) call. Then when you get the ping packet, you know your query has finished.

INPUTS

session - The session to send the get op to.

wildpath - The path of the data items you wish to retrieve.

You may include wildcards to retrieve multiple items.

maxBytes - The maximum number of bytes of data you wish to receive with each item. Larger data buffers will be truncated to this length. Since data items can be arbitrarily large, you can use this to avoid unpleasant surprises. Specifying -1 for this argument will allow any size data to be downloaded.

RESULTS

Returns the assigned ID number for the get operation on success, or 0 on failure. Data that is retrieved is sent asynchronously, as QMessages, to session->qMsgPort. A QGetOp that does not find any matching data items is not considered an error, so it is possible that no messages will be sent in response to this op.

EXAMPLE

```
LONG transID;
/* Get nodes in all clients' home directories named testNode */
if (transID = QGetOp(session, "/#?/#?/testNode", -1))
printf("Get op succeeded, was given id #%%li\n",transID);
else
printf("Get op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#) , [FreeQMessage](#) , [QSubscribeOp](#)

1.24 qdeleteop

amarquee.library/QDeleteOp amarquee.library/QDeleteOp

NAME

QDeleteOp - Delete the given set of entries from the Amarquee server.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QDeleteOp(struct QSession * session, char * wildpath)
```

FUNCTION

This function tells the AMarquee server to remove and free the entries stored in wildpath.

This function may also be used in direct client-to-client connections (as created via [QNewSession](#) and [QNewHostSession](#)), in which case the other client will receive a QMessage with

qm_Data equal to NULL.

NOTE

You may only remove entries in your own directory. Attempting to remove other people's data will result in you being sent a QERROR_UNPRIVILEGED error [QMessage](#) .

This function will also immediately delete any streaming buffers that were created for the given node(s) via [QStreamOp](#) .

To ensure that all stream updates were received, you may want to do a [QGo](#) (QGOF_SYNC) and wait for the returned sync QMessage before QDeleteOp'ing the node(s).

INPUTS

session - The session to send the delete op to.

wildpath - The path of the data items you wish to remove.

You may include wildcards to remove multiple items.

RESULTS

Returns the assigned ID number of the delete operation on success, or 0 on failure. Any server-side errors will be sent asynchronously as [QMessages](#) .

EXAMPLE

```
LONG transID;
if (transID = QDeleteOp(session, "myNode"))
printf("Delete op succeeded, was given id #%%li\n",transID);
else
printf("Delete op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#)

1.25 qrenameop

amarquee.library/QRenameOp amarquee.library/QRenameOp

NAME

QRenameOp - Rename a data entry to another name.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QRenameOp(struct QSession * session, char * path, char * label)
```

FUNCTION

This function tells the AMarquee server to rename a node to another label. Nodes may only be renamed in their current directory; they can not be moved to another directory with this command.

NOTE

Other clients will see this operation as a deletion of the old node and the creation of the new one.

Also note that while "path" should be a regular node path name, "label" should just be just the name of the node itself--the rest of the path is assumed to be the same.

You can only rename nodes in your own directory tree.

INPUTS

session - The session to send the rename op to.

path - The pathname of the node to rename

label - The new label the node is be renamed to.

RESULTS

Returns the assigned ID number of the rename operation on success, or 0 on failure. Any server-side errors will be sent asynchronously as [QMessages](#) .

EXAMPLE

```
LONG transID;
/* Changes node /myHost/myProgram/myDir/node1 into
/myHost/myProgram/myDir/node2 */
if (transID = QRenameOp(session, "myDir/node1", "node2"))
printf("Rename op succeeded, was given id #%%li\n",transID);
else
printf("Rename op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#)

1.26 qsubscribeop

amarquee.library/QSubscribeOp amarquee.library/QSubscribeOp

NAME

QSubscribeOp - Start watching the given data items for updates.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QSubscribeOp(struct QSession * session, char * wildpath, LONG maxBytes)
```

FUNCTION

This function tells the AMarquee server that your client has a continuing interest in the given data item(s) specified in "wildpath". When any items meeting the pattern specified in "wildpath" are created, changed, or deleted, you will be sent

QMessages reflecting their new contents.

(Nodes being deleted will result in you receiving a **QMessage** with `qm_Data` being `NULL`.)

NOTE

This function will not cause any data to be sent to you until a change in the data's state occurs. If you are just starting up your session and wish to get the full current state of the data, you should probably send a **QGetOp** as well.

INPUTS

`session` - The session to send the subscribe op to.

`wildpath` - The regular path name indicating which items you are interested in. Regular expressions are allowed.

`maxBytes` - Works similarly to the `maxBytes` argument in **QGetOp** .

If the data portion of entries sent to you is longer than "`maxBytes`", it will be truncated to `maxBytes` bytes. Set this to `-1` to allow any size data to be sent.

RESULTS

Returns the assigned ID number of the subscribe operation on success, or `0` on failure. Any server-side errors will be sent asynchronously as **QMessages** . Data **QMessages** resulting from this command may continue to be sent indefinitely, or until you cancel your subscription with **QClearSubscriptionOp** .

EXAMPLE

```
LONG transID;
/* Tells AMarquee to send us a message whenever a program
named ExampleProgram changes a node named data in its home dir */
if (transID = QSubscribeOp(session, "#?/ExampleProgram/data", -1))
printf("Subscribe op succeeded, was given id #%%li\n",transID);
else
printf("Subscribe op failed. (no memory?)\n");
```

SEE ALSO

QGo , **QClearSubscriptionOp**

1.27 qsetop

`amarquee.library/QSetOp` `amarquee.library/QSetOp`

NAME

QSetOp - Create or update a data item with a new buffer of data.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QSetOp(struct QSession * session, char * path, void * buffer, ULONG bufferLength)
```

FUNCTION

This function allows you to upload to the server data you wish to be made public. The data sent is stored as a simple buffer of bytes, and can thus be any data type you wish to send.

This function may also be used in direct client-to-client connections (as created via [QNewSession](#) and [QNewHostSession](#)), in which case the data in the arguments is received directly by the other client.

NOTE

You may only set nodes in your own directory.

This function is preferable to [QStreamOp](#) when the data you are posting is absolute, and it matters more that the other client programs see the latest revision of the data as soon as possible, than to have them always receive every update of the data. Also, this function is more memory-efficient than [QStreamOp](#) .

This operation will immediately clear any streaming data buffers that were previously in use for the given node. Thus, if you were previously using [QStreamOp](#) on a node and then want to do a [QSetOp](#) on that same node, it is a good idea to do a [QGo](#) (QGOF_SYNC) and wait for the sync [QMessage](#) first, before executing the [QSetOp](#), so that none of your streamed updates will be lost.

INPUTS

session - The session to wish to send the set op to.

path - The regular path of the node you wish to create or update. Wildcards are not allowed here.

buffer - A pointer to the first byte of the data buffer you wish to upload, or NULL if you wish to delete an existing node specified by "path".

bufferLength - The length of the data buffer, in bytes.

RESULTS

Returns the assigned ID number of the set operation on success, or 0 on failure. Any server-side errors will be sent asynchronously as [QMessages](#) .

EXAMPLE

```
LONG transID;
```

```
LONG data[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
/* Upload the data array into a node named "data" in our home dir */
if (transID = QSetOp(session, "data", data, sizeof(data)))
printf("Set op succeeded, was given id #%%li\\n",transID);
else
printf("Set op failed. (no memory?)\\n");
SEE ALSO
QGo , QStreamOp
```

1.28 qmessageop

amarquee.library/QMessageOp amarquee.library/QMessageOp

NAME

QMessageOp - Send a message to one or more other clients.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QMessageOp(struct QSession * session, char * hosts, void * buffer, ULONG bufferLength)
```

FUNCTION

This function allows you to send a buffer of data to other clients on the AMarquee server without permanently storing any information on the server. This can be more efficient than using [subscriptions](#) to transmit data, and it allows you to easily specify which clients should receive the message, on a per-message basis.

Your message will be sent to every host that matches the regular expression specified in the "hosts" argument, provided that that host has specified that it will accept messages from your client (via [QSetMessageAccessOp](#)). The regular expression in "hosts" should be of the form "/foo/bar".

If no other clients received your message (either because no clients matching your specification existed, or because they have not granted you message access), you will be receive a `QERROR_UNPRIVILEGED` `QMessage` notifying you of this.

When another client sends your client a `QMessageOp`, the `QMessage` you receive will have a `qm_ID` of zero, and the `qm_Path` field will contain the path of the home node of the message's sender (e.g. "/host/prog").

This function may also be used in direct client-to-client connections (as created via [QNewSession](#) and [QNewHostSession](#)), in which case the data in the arguments is received directly by the other client.

NOTE

By default, all clients have messaging access turned off. So in order to send a message to a client, that client must have first allowed it with [QSetMessageAccessOp](#) .

Data passed with this function will be streamed, so you do not have to worry about synchronization.

INPUTS

session - The session to wish to send the message op to.

hosts - A regular expression indicating which set of clients you wish to send the message to. It must be of the form `"/foo/bar"`. For example: to send your message to all clients named "Bob", do `"/#?/Bob"`.

buffer - A pointer to the first byte of the data buffer you wish to transmit. If NULL is specified, a one-byte buffer containing the NUL byte will be transmitted.

bufferLength - The length of the data buffer, in bytes.

RESULTS

Returns the assigned ID number of the message operation on success, or 0 on failure. Any server-side errors will be sent asynchronously as [QMessages](#) .

EXAMPLE

```
LONG transID;
```

```
LONG data[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
/* Send a message containing the array data to all programs named  
ExampleProgram. */
```

```
if (transID = QMessageOp(session, "/#?/ExampleProgram", data, sizeof(data)))
```

```
printf("Message op succeeded, was given id #%%li\n",transID);
```

```
else
```

```
printf("Message op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#) , [QSetMessageAccessOp](#)

1.29 qstreamop

amarquee.library/QStreamOp amarquee.library/QStreamOp

NAME

QStreamOp - Create or update a data item with a new buffer of data.

Use data streaming so that other clients will not miss any updates, even if you don't use any synchronization.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QStreamOp(struct QSession * session, char * path, void * buffer, ULONG bufferLength)
```

FUNCTION

This function works essentially the same as [QSetOp](#), except that you do not need to worry about data synchronization problems (e.g. changing the data in a node a second time before some clients had read the first value). Instead, the AMarquee server will ensure that all stream updates are seen by all interested clients, in the order they were sent.

In direct client-to-client connections, this function operates exactly the same as [QSetOp](#) does.

NOTE

You may only stream nodes in your own directory.

QStreamOp is less memory-efficient on the server side than QSetOp is, because the server may have to keep around multiple revisions of your data in the node you QStreamOp on.

This feature requires v38+ of amarquee.library, and v1.10B+ of AMarqueed.

INPUTS

session - The session to wish to send the stream op to.

path - The regular path of the node you wish to create or update. Wildcards are not allowed here.

buffer - A pointer to the first byte of the data buffer you wish to upload, or NULL if you wish to delete an existing node specified by "path".

bufferLength - The length of the data buffer, in bytes.

RESULTS

Returns the assigned ID number of the stream operation on success, or 0 on failure. Any server-side errors will be sent asynchronously as [QMessages](#).

EXAMPLE

```
LONG transID;
```

```
LONG data[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
/* Stream Upload the data array into a node named "data" in our home dir */
```

```
if (transID = QStreamOp(session, "data", data, sizeof(data)))
```

```
printf("Stream op succeeded, was given id #%%li\n",transID);
```

```
else
```

```
printf("Stream op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#), [QSetOp](#)

1.30 qclearsubscriptionop

amarquee.library/QClearSubscriptionOp amarquee.library/QClearSubscriptionOp

NAME

QClearSubscriptionOp - Remove a subscription or all your subscriptions

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QClearSubscriptionsOp(struct QSession * session, LONG which)
```

FUNCTION

This function may be used to tell the AMarquee server that you wish to remove one or all of your current subscriptions.

NOTE

No error will be returned, and no harm done, if you specify the deletion of a subscription id that you do not have in effect.

INPUTS

session - The session that you wish to send the clearsubscription op to.

which - The transaction ID of the subscription to clear (as was returned to you by [QSubscribeOp](#)), or 0 if you wish to clear all of your current subscriptions.

RESULTS

Returns the assigned ID number of the clearsubscription operation on success, or 0 on failure.

EXAMPLE

```
LONG transID;
```

```
/* clear all our previous subscriptions */
```

```
if (transID = QClearSubscriptionOp(session, 0))
```

```
printf("ClearSub op succeeded, was given id #%li\n",transID);
```

```
else
```

```
printf("ClearSub op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#) , [QSubscribeOp](#)

1.31 qpingop

amarquee.library/QPingOp amarquee.library/QPingOp

NAME

QPingOp - Requests that the Amarquee server respond with a "ping" message.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QPingOp(struct QSession * session)
```

FUNCTION

This function tells the AMarquee server to return a ping [QMessage](#) .

This can be useful if you wish to be sure that your AMarquee process has completed all previous operations you may have sent it.

This function may also be used in direct client-to-client connections (as created via [QNewSession](#) and [QNewHostSession](#)), in which case the other client will receive an [QMessage](#) containing how much memory is free on your machine.

NOTE

While this operation will help you synchronize to the completion of work by your **own** AMarqueued server process, it will not guarantee that by the time you receive your "ping" [QMessage](#) , that your changes have been propagated to all the **other** AMarqueued processes. You can use [QGo](#) ([QGOF_SYNC](#))'s ping reply to guarantee that.

INPUTS

session - The session to send the ping operation to.

RESULTS

Returns the assigned ID number of the ping operation or 0 to indicate failure. A ping [QMessage](#) will be returned asynchronously.

EXAMPLE

```
LONG transID;
/* Send a ping to the server, that will be sent back ASAP */
if (transID = QPingOp(session))
printf("Ping op succeeded, was given id #li\n",transID);
else
printf("Ping op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#) , [QInfoOp](#)

1.32 qinfoop

amarquee.library/QInfoOp amarquee.library/QInfoOp

NAME

[QInfoOp](#) - Request that the AMarquee server respond with an "info ping" message.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

LONG QInfoOp(struct QSession * session)

FUNCTION

Works essentially the same as [QPingOp](#) , only the returned "ping" QMessage will have a struct QRunInfo as data, reflecting the current memory state of the AMarquee server.

This function may also be used in direct client-to-client connections (as created via [QNewSession](#) and [QNewHostSession](#)), in which case the other client will receive an info QMessage.

NOTE

Of course, any memory state info sent to you by the AMarquee server is very likely to be out of date by the time you get it!

Also, when using the contents of QRunInfo to estimate how much information you can safely store on the AMarquee server, it will be helpful to know that the server makes a copy of everything it receives before deleting the original data, so qr_Avail should be AT LEAST 2 times the amount you wish to upload, and probably 3 or 4 times is safer.

Lastly, the QRunInfo struct will likely expand (in a compatible way) in future versions of AMarquee.

INPUTS

session - The session to send the info operation to.

RESULTS

Returns the assigned ID number of the ping operation or 0 to indicate failure. A ping [QMessage](#) (with a QRunInfo struct in the qm_Data field) will be returned asynchronously.

The returned QRunInfo struct (defined in AMarquee.h) is as follows:

```
struct QRunInfo
{
    LONG qr_Allowed; /* The theoretical maximum number of bytes your server may allocate. */
    LONG qr_Allocated; /* The number of bytes currently allocated by your server. */
    LONG qr_Avail; /* The number of bytes that may actually be allocated by your server. */
};
```

qr_Allowed will be constant throughout your AMarquee session.

qr_Allocated will depend solely on your program's actions.

qr_Avail depends on qr_Allowed, qr_Avail, and also the amount of free memory available on the server computer. It may change at any time.

EXAMPLE

```
LONG transID;
```

```
/* Send a ping to the server, that will be sent back ASAP */
if (transID = QInfoOp(session))
printf("Info op succeeded, was given id #%%li\n",transID);
else
printf("Info op failed. (no memory?)\n");
SEE ALSO
QGo , QPingOp
```

1.33 qsetaccessop

amarquee.library/QSetAccessOp amarquee.library/QSetAccessOp
NAME

QSetAccessOp - Set a path describing which other clients may
access your data.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QSetAccessOp(struct QSession * session, char * newAccess)
```

FUNCTION

By default, any other AMarquee client may download your client's
data. But it may be that you only want to share your data with
certain clients. This function allows you to set an access
control path, in the form of "/hostExp/progExp", to specify
exactly which other AMarquee clients may look at your data.
Clients that are not included in your access path will not
be notified when your data is updated, and they will not see
any data in your directory via QGetOp(), either.

NOTE

While you can hide your data from other clients, you cannot
hide your presence. Unauthorized clients will still be able
to look at your root node, and may still be notified when your
session begins or ends.

If you specify an access pattern that excludes your own data(!),
you will not be able to read your data using global node paths
(e.g. QGetOp ("/myhost/myprog/mydata") will not return any QMessages
to you), but you can still read your data using local node paths
(e.g. QGetOp ("mydata"), which means the same thing, will work).

You can always QSetOp to your directory, no matter what.

INPUTS

session - The session you wish to send the access operation to.

`newAccess` - The new access pattern to use. Note that on startup, the access pattern is `"#?/#?"` (i.e. no restrictions on access).

RESULTS

Returns the assigned ID number of the access operation, or 0 to indicate failure. This function will fail and return 0 if the `"newAccess"` arg is not in the form `"/foo/bar"`.

EXAMPLE

LONG transID;

```
/* Let only programs named ExampleProgram see our data */
if (transID = QSetAccessOp(session, "#?/ExampleProgram"))
printf("SetAccess op succeeded, was given id #%li\n",transID);
else
printf("SetAccess op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#) , [QGetOp](#) , [QSubscribeOp](#)

1.34 qsetmessageaccessop

amarquee.library/QSetMessageAccessOp amarquee.library/QSetMessageAccessOp

NAME

`QSetMessageAccessOp` - Specify which other clients may send you messages with [QMessageOp](#) .

SYNOPSIS

```
#include <clib/amarquee_protos.h>
```

```
LONG QSetMessageAccessOp(struct QSession * session, char * newAccess, LONG maxBytes)
```

FUNCTION

This function allows you to specify which other clients may send you data directly using the [QMessageOp](#) function.

NOTE

When you first connect, messaging access is turned off, and no other client may send you messages. (This ensures that you do not receive `QMessages` that you were not expecting!)

INPUTS

`session` - The session you wish to send the message access op to.

`newAccess` - The new access pattern to use, or `NULL` if you wish to allow no messages to be sent to your program.

`maxBytes` - The maximum size that any data buffer sent to your client may be. Messages longer than `"maxBytes"` bytes

long will be truncated before they are sent to you.

If you specify maxBytes as -1, no restriction will be placed on message length.

RESULTS

Returns the assigned ID number of the access operation, or 0 to indicate failure. This function will fail and return 0 if the "newAccess" arg is not in the form "/foo/bar".

EXAMPLE

```
LONG transID;
/* Let programs named ExampleProgram send us messages */
if (transID = QMessageSetAccessOp(session, "/#?/ExampleProgram"))
printf("MessageSetAccess op succeeded, was given id %li\n",transID);
else
printf("MessageSetAccess op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#) , [QMessage](#)

1.35 freeqmessage

amarquee.library/FreeQMessage amarquee.library/FreeQMessage

NAME

FreeQMessage - Frees the given QMessage and its associated data.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
void FreeQMessage(struct QSession * session, struct QMessage * qmsg)
```

FUNCTION

Frees the given QMessage and its associated data. You must call this function on EVERY [QMessage](#) that you receive, sometime before you [QFreeSession](#) the session that you got it from. After you [FreeQMessage](#) a QMessage, you may no longer access any of the data it contained. Your app will be more memory efficient if you free your QMessages as soon as possible.

NOTE

QMessages use a special memory-sharing scheme, so you MUST use this function and not FreeMem() or ReplyMsg() or anything else on QMessages!

INPUTS

session - The session from which you received the QMessage
qmsg - The QMessage to free.

RESULTS

None.

EXAMPLE

```
struct QSession * s;
struct QMessage * qmsg;
/* ... setup QSession s ... */
while(qmsg = GetMsg(s->qMsgPort->mp_SigBit))
{
/* ... process qmsg ... */
FreeQMessage(qmsg);
}
```

SEE ALSO

[QMessages](#)

1.36 qgo

amarquee.library/QGo amarquee.library/QGo

NAME

QGo - Instructs the TCP thread to begin transmitting the currently accumulated transaction packet.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
LONG QGo(struct QSession * session, ULONG flags)
```

FUNCTION

All of the Q#?Op() functions in amarquee.library create transaction packets which are queued on the client computer. That is, they are not immediately transmitted to the AMarquee server. Rather, when you wish to transmit the operations you have accumulated, you should call this function.

NOTE

The QGOF_SYNC option of this command is different from doing a [QPingOp](#), because with this option you are guaranteed that all other server processes have seen your new data by the time you get the back ping packet that this operation generated.

INPUTS

session - The session you wish to begin transmission.
 flags - A bit chord of the QGOF_* flags defined in AMarquee.h
 Flags currently supported are:

QGOF_SYNC - If set, the AMarquee server will be return a ping packet after all of the operations currently queued have executed, and all other AMarquee server threads have been notified of any updates made. The ping packet will have a qm_ID equal to the return value of this function.

RESULTS

Returns the assigned ID number of the QGo operation or 0 to indicate failure.

EXAMPLE

```
LONG goID;
/* ... do various Q*Op() calls ... */
if (goID = QGo(OL))
printf("Go succeeded, was given id #li\n",goID);
else
printf("Go failed. (no memory?)\n");
```

SEE ALSO

[QMessages](#)

1.37 installdaemon

Note: These are instructions for installing a daemon program written using amarquee.library, NOT for installing the "AMarqueed" program itself! For instructions on how to do that, see [here](#) .

There are several steps involved in installing an AmiTCP daemon that will be activated by inetd.

First, you must choose a name for the service the daemon will provide. In this example, we'll call the service AMarqueeServer, because that is the name of the daemon demo included with this archive. But it could be any name.

Second, you must choose which port the service will reside on. For this example, I'll choose port 16000, but it can be almost any port number, as long as no other service is using that same number.

Now you have to edit a couple of config files.

First, open up amitcp:db/services in your text editor and add a line at the bottom for your new service. For amarquee.library based servers, that line should have the format:

ServiceName portNumber/tcp

So in our example, add the line:

```
AMarqueeServer 16000/tcp
```

Now save the services file and open the file amitcp:db/inetd.conf.

Again, we want to add a line for our service at the bottom of the file. For amarquee-based servers, that line should have this format:

```
ServiceName stream tcp nowait root fullPathNameToExecutable
```

So for our example, add the line:

```
AMarqueeServer stream tcp nowait root amitcp:serv/AMarqueeServer
```

Don't forget to copy the executable file "AMarqueeServer" into the amitcp:serv directory, or change the last field of the above config line to reflect the executable's real location on your disk!

Now, after you re-start AmiTCP, the AMarqueeServer service should be available on port 16000. To make sure it works, you can try to connect to it with the AMarqueeDebug utility:

```
AMarqueeDebug localhost JustAQuickTest 16000
```

And AMarqueeServer will open a window. If it doesn't seem to be working, make sure you have AmiTCP configured so that the inetd program is running, and use SnoopDos to see if your executable file is being loaded correctly.

1.38 exampleclients

I have written and included some little AMarquee toy applications in this archive. These applets come with C source code, and are meant as examples on how to use AMarquee. They aren't as robust or useful as they could be, but they do demonstrate some possible AMarquee uses and techniques.

Applications that can currently be found in the "Examples" directory:

AMarqueeDebug - A cli interface to AMarquee. Connects to an AMarquee server on the server and port specified on the command line. Once connected, it allows you to execute various AMarquee commands and see incoming QMessages.

AMarqueeHost - Like AMarqueeDebug, only instead of initiating a connection, it uses [QNewHostSession](#) to await an incoming connection on a specified port.

AMarqueeServer - Something like AMarqueeHost, only implemented as an

AmiTCP daemon using [QNewServerSession](#) . To start this program, you should [install](#) it in inetd's service registry, and then connect to it via an AMarquee client.

For example, if you installed it on port 16000, you could connect to AMarqueeServer using AMarqueeDebug:

```
AMarqueeDebug localhost test 16000
```

BounceCount - Tests the data subscription. Subscribes to `"/#?/#?/count"`, and whenever it receives a message that someone else has updated their count to a higher value than its own, it responds by updating its value one further.

MiniIRC - A primitive IRC system. Just one "room", but hey--it's only 212 lines of C source code!

RemoveTest - Tests the node removal notification by continuously creating a node and then deleting it. You can use AMarqueeDebug or whatever to subscribe to `"/#?/#?/switch"` to see that it works.

SillyGame - A trivial multiplayer "game" implemented with AMarquee. You can use the numeric keypad to move your letter around, or press any other key to change to another letter. Now uses data streaming for better performance!

SyncTest - Tests the QGo() synchronization. Anyone who subscribes to `"/#?/#?/count"` should see its value increasing without ever missing a step.

1.39 Thanks

Thanks go to the following people:

Oliver Hotz, for hosting the first full-time AMarqueed server site (at isys.ml.org, port 2957).

Ryan Ojakian and Trina Arth... two incredible human beings who know what "mahuhmanah" means! :)

Meni Berman, for his help beta-testing.

Håkan Parting and Markus Lamers for their alerting me to the existence of evil bugs in `amarquee.library`.

Fredrik Rambris, for providing the Miami portion of the Installer script.

The good folks on `comp.sys.amiga.programmer`, for suggesting solutions to several show-stopping programming problems with the code.

1.40 faq

Q: What is a good AMarquee server to connect to?

A: Right now there is just isys.ml.org, which is the server for QAmiTrack. You can also use it for other programs, for now, but if there is too much load on it, the site administrator might have to restrict it to just QAmiTrack use, so go easy! :) Hopefully, other sites will appear for use with other programs...

Q: What cool AMarquee programs are there?

A: Well, at the time of this writing there is QAmiTrack and ARemote, written by me, and AmiComSys, by Håkan Parting. All are available on Aminet. Other apps are coming!

A2: Oh yeah, if you're bored you can check out the example programs included with this archive. SillyGame is my personal favorite. :)

Q: Why can't I connect to an AMarquee server?

A: There could be any of a number of reasons:

- The Amiga running the server is down or inaccessible.
- The Amiga running the server has banned your site or client program, or has reached its maximum number of connections allowed.
- The Amiga running the server is running low on memory.

1.41 The ground was littered with squashed bugs...

("-" = new feature, "*" = bug fix)

1.43 : (Public Release 4/27/97) (amarquee.library v43)

* amarquee.library's QMessage freeing system had a design flaw that was causing Enforcer hits and memory leaks. Fixed that, and as a side effect, [FreeQMessage](#) is now much more efficient.

* Fixed a bug in [QFreeSession](#) that would cause it to deadlock and hang the calling process when freeing QSessions that were allocated with [QNewHostSession](#) .

* Fixed some typos, layout errors, and anachronisms in the .guide file.

1.41 : (Public Release 4/17/97) (amarquee.library v42)

* Fixed a nasty race condition in amarquee.library that could cause crashes and/or memory leaks if the priority of the TCP client thread was different from that of the user's thread.

Thanks to Håkan Parting and Markus Lamers for reporting this bug!

1.40 : (Public Release 4/8/97) (amarquee.library v41)

- Added `QNewSessionAsync` to amarquee.library.

Now you can connect without temporarily freezing up your GUI!

- * The Installer script now puts "resident AMarquee pure" into the user-startup, so that resindenting will still work even if AMarqueed's PURE bit isn't set.

1.30β : (Public Beta Release 2/9/97) (amarquee.library v40)

- Both AMarqueed and amarquee.library flush their TCP output buffer after each transaction group has finished being queued. This causes the packets to be sent sooner, allowing for faster response times.

- Added code to SillyGame that syncs with the AMarquee server on exit, so that the player's marker will disappear from the other clients' SillyGame windows when he exits.

- Changed the behavior of QSessions created by `QNewHostSession`.

Now transactions sent to them while they are still unconnected will cause `QERROR_NO_CONNECTION` QMessages to be returned.

- Changed `QNewHostSession()` to allow automatic port selection by the TCP stack.

- The Installer script now supports Miami. (Thanks to Fredrik Rambris for providing this)

- Added an example section to all the man pages in the docs.

- * Rewrote the client TCP thread shutdown code to be synchronous.

Before, the library sent a signal to cause the TCP thread to quit, and this could cause the last transaction to be dropped. I think this was the bug that was causing SillyGame to crash occasionally.

1.20β : (Public Beta Release 2/4/97) (amarquee.library v39)

- Changed AMarqueed's behavior when a hostName/progName duplicate occurs. Before, the new connection was denied. Now, the old connection quits to make room for the new one, and the new connection proceeds normally.

- Added an idle-time ping/timeout capability to AMarqueed, so that dead clients are detected and removed in a reasonable amount of time. Also added the `AMARQUEED_PINGRATE` env variable option.

- * Bumped amarquee.library's revision number to v39 (it was at v37 in both release 1.10β and 1.00β)

- * Amarquee TCP handling threads now do the right thing when AmiTCP is shutting down (they send a `QERROR_NO_CONNECTION`

to the user program and close SocketBase).

1.10β : (Public Beta Release 1/28/97) (amarquee.library v38)

- Added the **QStreamOp** function to the **amarquee API** .
- Added the **QMessageOp** function to the **amarquee.library API** .
- Added the **QSetMessageAccessOp** function to the **amarquee.library API** .
- Added the **QNewServerSession** function to the **amarquee API** .

Now you can use amarquee.library to make AmiTCP/inetd style server programs.

- Added the **QNewHostSession** function to the **amarquee API** .

Now you can connect directly to other amarquee clients if you wish, rather than sending all data through the server.

- Added the **AMARQUEED_PRIORITY** env variable option.
- Added the **AMARQUEED_FAKECLIENT** env variable option.
- Rewrote AMarqueued to support streaming data.

1.00β : (Public Beta Release 01/15/96) (amarquee.library v37)

- First beta release.

1.42 What's Next?

Note: These are things I'm thinking of implementing; Whether I actually implement them or not depends on how difficult they will be to implement and user response (both in the form of **communications** and **donations**).

- Fix bugs, fix bugs, fix bugs!
- An ARexx version of amarquee.library?
- Server linking?
- Write some decent AMarquee client programs.

1.43 unnamed.1

1.44 Known Bugs and Other Problems

- Some memory leaks?
- Sometimes when I run a few AMarquee sessions and then run "Amiga System Probe" to check the state of the system out, the ASP window never opens, I can no longer use ARexx or open or close windows. This may be a bug in AMarquee, or in ASP, I'm not sure.
- Despite the data-truncation facilities provided by

the maxBytes arguments in various functions, evil people could still flood your data path, either by sending you lots of little messages or by sending messages with very long pathNames...

- An apparent bug in AmiTCP3.0b2 can cause the AMarquee daemon program to emit the following Enforce hits if it is transmitting a packet when the client closes the connection. Miami does not have this problem, and later versions of AmiTCP have not been tested. The Enforcer hits look like:

```
LONG-READ from 0000001C PC: 07BFA0E4
```

```
USP: 07D9EE78 SR: 0000 SW: 0749 (U0)(-)(-) TCB: 07D97198
```

```
Name: "AMarquee [127.0.0.1]" CLI: " "
```

```
BYTE-READ from 0000001B PC: 07BFA12A
```

```
USP: 07D9EE78 SR: 0000 SW: 0751 (U0)(-)(-) TCB: 07D97198
```

```
Name: "AMarquee [127.0.0.1]" CLI: " "
```

```
BYTE-WRITE to 0000001B data=04 PC: 07BFA130
```

```
USP: 07D9EE78 SR: 0004 SW: 0711 (U0)(-)(-) TCB: 07D97198
```

```
Name: "AMarquee [127.0.0.1]" CLI: " "
```

1.45 otherprogs

Other Amiga programs I have written (all require AmigaDos2.04 or higher):

GadMget - Loads in an Aminet RECENT or INDEX file and lets you choose files to download via a pair of ListViews. Features keyword searching and sorting by name, size, age, directory, and description. When you're done, it outputs the ftp commands that are needed to download the selected files. The output formatting is extremely flexible, allowing generation of many formats: ftp, ncftp, ftp-by-mail, shell scripts, etc.

Comes with an ARexx script to completely automate downloading with ncFTP. (util/misc/GadMget2.05.lha,93K)

AmiSlate - A paint program that works with AmiTCP to allow two people to cooperatively paint on the same drawing from different computers. Features an extensive ARexx port which allows the construction of new features and games. Comes with ARexx scripts for chess, tic-tac-toe, backgammon, and others.

(comm/tcp/AmiSlate1.4.lha,115K)

QAmiTrack - An AMarquee program that lets Amigans find each other on the net. (comm/net/QAmiTrack1.80.lha)

ARemote - An AMarquee program that allows you to transmit your mouse movements and keystrokes over a TCP connection, so that you can control all your Amigas from one keyboard.

(comm/net/ARemote1.00B.lha)

AmiPhone - An Internet voice-chat program, similar to IPhone and VoiceChat and Nevot and all that, only Amiga-specific.

Features a flexible buffering mechanism for slow connections, an ARexx port, IFF 8SVX transmission and playback, internal multitasking, and support for a variety of digitizers.

(comm/net/AmiPhone1.92.lha,142K)
