

AMarquee

Jeremy Friesner

COLLABORATORS

	<i>TITLE :</i> AMarquee		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Jeremy Friesner	February 12, 2023	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	AMarquee	1
1.1	Contents	1
1.2	miami	2
1.3	residentnote	3
1.4	Installation	3
1.5	amitcpinstall	4
1.6	inet225install	4
1.7	credits	5
1.8	How to reach me	5
1.9	disclaimer	5
1.10	AMarquee is DonationWare	5
1.11	What you need	7
1.12	introduction	8
1.13	Using AMarqueed	9
1.14	Using amarquee.library functions from ARexx scripts	12
1.15	Using amarquee.library	13
1.16	sessionclass	14
1.17	api	15
1.18	usingintro	16
1.19	qmessages	17
1.20	qnewsession	20
1.21	qnewsessionasync	22
1.22	qnewhostsession	24
1.23	qnewserversession	27
1.24	qfreesession	29
1.25	qdetachsession	30
1.26	qreattachment	33
1.27	qnumqueuedpackets	35
1.28	qerrorname	36
1.29	qnumqueuedbytes	36

1.30	qdebugop	37
1.31	qgetop	38
1.32	qdeleteop	40
1.33	qrenameop	41
1.34	qsubscribeop	42
1.35	qgetandsubscribeop	43
1.36	qsetop	45
1.37	qmessageop	46
1.38	qsysmessageop	48
1.39	qstreamop	50
1.40	qclearsubscriptionop	51
1.41	qpingop	52
1.42	qinfoop	53
1.43	qsetaccessop	55
1.44	qsetmessageaccessop	56
1.45	qrequestprivilegesop	57
1.46	qreleaseprivilegesop	58
1.47	qkillclientsop	59
1.48	getnextqmessage	61
1.49	getqmessagefield	62
1.50	paramprivs	63
1.51	qsetparameterop	65
1.52	qgetparameterop	66
1.53	freeqmessage	67
1.54	qgo	68
1.55	installdaemon	70
1.56	exampleclients	71
1.57	Thanks	73
1.58	faq	73
1.59	The ground was littered with squashed bugs...	74
1.60	What's Next?	78
1.61	unnamed.1	79
1.62	Known Bugs and Other Problems	79
1.63	otherprogs	80

Chapter 1

AMarquee

1.1 Contents

AMarquee V1.47 by **Jeremy Friesner**

AMarquee is a mechanism by which Amiga programs and ARexx scripts on multiple Amigas may easily communicate data to each other in a many-to-many, "broadcast" fashion. It consists of two parts, an inetd daemon for the server computer to run, and a shared library for client computers to run. 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 work:Miami/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. (Note that you should change "work:Miami/AMarqueed" to reflect the actual location of the AMarqueed file!)
- 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 AMarqueued executable be run when an AMarquee client connects to your computer: resident, and non-resident. If the AMarqueued 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 AMarqueued 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. You may need to change the path above to accurately reflect the location of the AMarqueued executable on your system. (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 or inet):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, Inet225, or Miami, you can use the included Installer script to install AMarquee (actually, you can use the Installer script to install everything but AMarqueued even if you are not using one of the above packages!).

If you don't like to use Installer scripts, there are manual installation instructions available for [AmiTCP](#) , [Inet225](#) , 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, AMarqueued, 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 AMarqueued 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/AMarqueued
Read this note about making AMarqueued 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 inet225install

AMarquee is made of two parts, each of which may be installed independantly of the other. The first part, amarquee.library.inet225, 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, then renaming it to "amarquee.library".

The second part, AMarqueued, 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 Inet225, do the following:

- 1) Copy the file AMarqueued.inet225 from the distribution into your inet:serv directory, and rename it to "AMarqueued".
- 2) Add the following line to the end of your inet:db/services file:
AMarquee 2957/tcp

3) Add the following line to the end of your inet:db/inetd.conf file:

```
AMarquee stream tcp nowait inet:serv/AMarqueed
```

[Read this note about making AMarqueed resident](#)

4) Re-start Inet225. Your AMarquee server should now be ready to accept connections. Try running one of the included [example](#) programs to "localhost" to test it out.

1.7 credits

AMarquee V1.47

Created by [Jeremy Friesner](#)

Compiled with DICE C by Matt Dillon

1.8 How to reach me

Here are some ways to get in touch with me:

by EMail: jfriesne@ucsd.edu

jaf@chem.ucsd.edu

jaf@praja.com

by SMail: Jeremy Friesner

9481 Questa Pointe

San Diego, CA 92126

1.9 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.

1.10 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 'AMarquee1.47.lha':

Original Packed Ratio Date Time Name

```
-----  
1233 595 51.7% 20-Feb-98 20:12:42 AMarquee.info  
163183 46524 71.4% 20-Feb-98 20:12:46 +amarquee.guide  
1542 1096 28.9% 20-Feb-98 20:12:46 +AMarquee.guide.info  
32196 16945 47.3% 20-Feb-98 20:12:44 +amarquee.library  
32344 17015 47.3% 20-Feb-98 20:12:44 +amarquee.library.inet225  
5209 2330 55.2% 20-Feb-98 20:12:40 +AMarquee.readme  
835 268 67.9% 20-Feb-98 20:12:42 +AMarquee.readme.info  
40072 21107 47.3% 20-Feb-98 20:12:42 +AMarqueed  
40104 21114 47.3% 20-Feb-98 20:12:42 +AMarqueed.inet225  
4123 1604 61.0% 20-Feb-98 20:12:46 +EditTextFile.rexx  
10976 6286 42.7% 20-Feb-98 20:12:50 +AMarqueeDebug  
7166 2349 67.2% 20-Feb-98 20:12:48 +amarqueedebug.c  
6753 2323 65.6% 20-Feb-98 20:12:48 +amarqueedebug.cpp  
7085 2319 67.2% 20-Feb-98 20:12:54 +amarqueedebug.rexx  
12920 7206 44.2% 20-Feb-98 20:12:50 +AMarqueeDebugMultiThread  
9537 3191 66.5% 20-Feb-98 20:12:48 +AMarqueeDebugMultiThread.c  
9200 5495 40.2% 20-Feb-98 20:12:50 +AMarqueeHost  
4175 1563 62.5% 20-Feb-98 20:12:52 +amarqueehost.c  
6380 2123 66.7% 20-Feb-98 20:12:48 +amarqueehost.rexx  
11868 6923 41.6% 20-Feb-98 20:12:50 +AMarqueeServer  
5307 1924 63.7% 20-Feb-98 20:12:50 +AMarqueeServer.c  
7400 4731 36.0% 20-Feb-98 20:12:48 +BounceCount  
2910 1167 59.8% 20-Feb-98 20:12:52 +BounceCount.c  
522 275 47.3% 20-Feb-98 20:12:52 +dmakefile  
494 262 46.9% 20-Feb-98 20:12:52 +dmakefile.bak  
2439 1114 54.3% 20-Feb-98 20:12:48 +killclients.rexx
```

10408 6410 38.4% 20-Feb-98 20:12:50 +MiniIRC
6379 2171 65.9% 20-Feb-98 20:12:54 +MiniIRC.c
1139 502 55.9% 20-Feb-98 20:12:52 +PascalTest.p
6984 4444 36.3% 20-Feb-98 20:12:48 +RemoveTest
2546 1035 59.3% 20-Feb-98 20:12:52 +RemoveTest.c
9324 5734 38.5% 20-Feb-98 20:12:50 +SillyGame
8214 2586 68.5% 20-Feb-98 20:12:52 +SillyGame.c
7404 4711 36.3% 20-Feb-98 20:12:50 +StreamCheck
3132 1242 60.3% 20-Feb-98 20:12:52 +StreamCheck.c
7276 4628 36.3% 20-Feb-98 20:12:50 +StreamGen
2612 1090 58.2% 20-Feb-98 20:12:52 +streamgen.c
7456 4717 36.7% 20-Feb-98 20:12:48 +SyncTest
3250 1302 59.9% 20-Feb-98 20:12:54 +SyncTest.c
2658 1181 55.5% 20-Feb-98 20:12:48 +sysmessage.rexx
3437 878 74.4% 20-Feb-98 20:12:56 +AMarquee_protos.h
1313 449 65.8% 20-Feb-98 20:12:56 +amarquee.fd
4161 1701 59.1% 20-Feb-98 20:12:56 +AMarquee.h
1881 513 72.7% 20-Feb-98 20:12:56 +AMarquee_pragmas.h
10614 2704 74.5% 20-Feb-98 20:12:54 +Session.h
24116 5957 75.2% 20-Feb-98 20:12:46 +Install_AMarquee
612 329 46.2% 20-Feb-98 20:12:46 +Install_AMarquee.info
7307 2124 70.9% 20-Feb-98 20:12:56 +AMarquee.i
4232 602 85.7% 20-Feb-98 20:12:58 +AMarquee.lib

562428 234859 58.2% 20-Feb-98 20:13:32 49 files

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

1.11 What you need

- Kickstart V37 (WorkBench 2.04) or higher
- Inet225, or AmiTCP3.0b2+ (or an AmiTCP compatible TCP stack, such as Miami, or TermiteTCP)
- ARexx, PCQ Pascal, or a C or C++ compiler, or even just an assembler, if you wish to write AMarquee programs.

1.12 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 Amarqueued server code are re-entrant, to minimize memory usage.

1.13 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 AMarqueed 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 AMarqueed 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 AMarqueed connections allowed. For example, entering "setenv AMARQUEED_TOTALMAXCONN 5" will ensure that there are never more than 5 AMarqueed processes running at once.

- AMARQUEED_PRIORITY

If you wish the AMarqueed 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, AMarqueed 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, AMarqueed

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.

- AMARQUEED_MAXQUEUEDMESSAGES

This option allows you to set a limit on the number of (internal) messages that an AMarqueued client may have pending at any one time. If an AMarqueued daemon exceeds this limit, it will immediately quit and disconnect its client. This is useful in situations where some clients are connecting over TCP connections that are too slow or lossy to carry the amount of data they are being sent by other clients. When this happens, their server daemon's outgoing message queue can grow quite large, unless you limit it with this variable. The default setting is unlimited. If you do set this variable, it's a good idea to set it to at least 50, so that clients aren't knocked off too easily.

=====

====

==== PRIVILEGE SETTINGS ====

====

Below is the list of environment variables that are used to specify which clients are allowed which privileges. You can use these to specify how much access any given computer or client type may have on your server.

All of these variables should be set to values of the same format. The format looks like this:

/hostExp/progExp

Where hostExp is a standard Amiga wildcard expression designating an Internet hostname (or group of hostnames), and progExp is a standard Amiga wildcard expression designating a client name (or group of client names). This format can be used to designate arbitrarily complex groups of connection types. Here are some examples of how this format can be used:

/#?.com/QAmiTrack -- means all connections coming from hostnames ending in .com, where the program is logging in with the ID "QAmiTrack".

/evil.hackers.com/#? -- denotes all connections coming from host evil.hackers.com (no matter what the login name of the program is)

/#?/~(EvilApp) -- specifies connections from any computer, as long as the client isn't logging in as "EvilApp"

/(computer1.ucsd.edulcomputer2.ucsd.edulcomputer3.ucsd.edu)/#?

-- This specifies only connections coming from the hosts computer1.ucsd.edu, computer2.ucsd.edu, or computer3.ucsd.edu

Note the initial '/' character! It is required.

Below are the environment variables that may be set as described above.

- AMARQUEED_BANNED

With this parameter, you can indicate which clients should not be allowed to connect to your AMarquee daemon at all. (For example, a friend of mine was getting too much Netris traffic on his server, so he did a "setenv AMARQUEED_BANNED /#?/#?Netris" to ban Netris from his machine.

If one BANNED variable isn't enough for you, v1.46+ of AMarquee will also look at the env variables AMARQUEED_BANNED0, AMARQUEED_BANNED1, ..., AMARQUEED_BANNED9. Clients that match any of the BANNED env

variables will not be allowed to connect.

- AMARQUEED_KILLCLIENTS

An AMarquee server is a law-and-order sort of place, and clients are discouraged from doing mean things to other clients. Sometimes, however, a little cruelty is justified and a client needs to die.

This variable lets you give some clients a licence to **kill** the server daemons, and hence the AMarquee connections of other clients.

- AMARQUEED_SENDSYSMESSAGES

New in v1.45 of AMarquee is the ability to send "system messages".

System messages are a special type of message, in that they should consist of ASCII text (256 chars or less), and may be sent to any client at any time (as long as that client has **requested** the QPRIV_GETSYSMESSAGES privilege). They are to be used to notify users of server events (such as imminent shutdown, etc). This variable lets you specify which clients may **send system messages** .

- AMARQUEED_ADMIN

This env variable lets you specify which clients may dynamically reconfigure the AMarquee server, by remotely setting many of the AMARQUEED_* environment variables. This is a very powerful privilege, since it can be used to gain all other privileges!

There are some **restrictions** on what clients with AMARQUEED_ADMIN access can do, but nonetheless you should be very cautious in giving out this privilege!

- AMARQUEED_ALLPRIVILEGES

Specifying a client under this variable is the same as specifying that client in ALL the above variables. This one immediately gives its clients EVERY PRIVILEGE! So be careful with this one, too!

1.14 Using amarquee.library functions from ARexx scripts

As of version 46, the functions within amarquee.library are accessible from ARexx scripts. To make amarquee.library's functions accessible to your ARexx script, you should add the following line to the beginning of your script:

```
call addlib('amarquee.library', 0, -204, 46)
```

(Note that the second argument **MUST** be -204, and **NOT** -30 like it is with other libraries! The third argument is the minimum version number; you should require at least amarquee.library

v46, as earlier versions have no ARexx support)

Once this call returns, you can use `amarquee.library`'s **functions** pretty much the same way as you would within a C program.

All arguments are passed and all results returned in the standard ARexx manner, as strings.

There are some minor differences in usage between the C and ARexx versions of the functions (mostly the difference is that some arguments are optional under ARexx). These are noted in the AREXX NOTES section of the man page for each function.

There are also two new functions that are available only via ARexx:

GetQMessageField (message, fieldname)

GetNextQMessage (session, timeout)

These are used for doing things that are easy in C but harder to do in ARexx.

Note that you must be careful when using `amarquee.library` from ARexx! In particular, you must ensure that **QFreeSession** is called on ALL active QSession's before your script exits!

Otherwise you risk crashing. Also, all returned QMessages must have **FreeQMessage** called on them, or you will leak memory.

See the ARexx scripts in the TestPrograms subdirectory for examples of how to use `amarquee.library` from ARexx.

1.15 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](#)

[The C++ wrapper class](#)

[Using `amarquee.library` from ARexx scripts](#)

1.16 sessionclass

Included with the Amarquee distribution is a C++ "wrapper" class that you can use if you wish, instead of invoking the amarquee.library calls directly.

Amarquee.library usage maps nicely to a C++ object:

- Using amarquee.library directly, you allocate a struct QSession, call functions to which you pass a pointer to your QSession struct, then call **QFreeSession** () to free the QSession struct when you are done.
- Using the Session class, you create a new Session object, call methods on the Session object, then delete the object when you are done.

Advantages of using the Session class over the straight C API include:

- Const correctness. The Session class API uses const where appropriate.
- Convenient default arguments are implemented where appropriate. For example, the C call QGo(qsession, 0L) is can be written in as session->go(), saving you a few keystrokes.
- You can put Session objects on the stack, and be sure that they are properly allocated and freed at the correct times.
- No performance penalty. All Session methods are inline, so using a Session is really no slower than calling the C functions directly.
- Since all the code needed to implement the Session class is included in Session.h, it is easy to modify the Session class to suit your tastes.

Disadvantages of the Session class:

- It requires a C++ compiler.
- It hasn't been run-tested. (This is because I don't have a C++ compiler on my Amiga) Nonetheless, I am fairly confident it is bug free, and if you look in Session.h you'll see why-- almost all the methods are one-line pass-thrus to the C API.

Exception throwing in the Session class:

- Exceptions are only thrown by one method in the Session class-- the class constructor. All other methods register failure (where applicable) by returning the same return codes that the C API does. The class constructor will throw a SessionCreationFailedException

(also defined in Session.h) if it cannot allocate the internally held QSession struct.

- If you are like me and don't like using exceptions in C++, there are static methods in the Session class that you can use to create instead of the Session constructor to allocate a Session object. These methods are named very similarly to their C API counterparts, and work the same way: They return a pointer to a newly allocated Session object on success, or NULL on failure. See Session.h for more info. Note that you CANNOT cast a Session to a (struct QSession), nor can you cast a (Session *) to a (struct QSession *). A Session object holds a (struct QSession *), but do not derive from it. Note also that you must still open and close amarquee.library (v47+) yourself--the Session class does not do it for you.

1.17 api

Library functions for both C and ARexx:

```
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)
LONG 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 QGetAndSubscribeOp (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 QSysMessageOp (struct QSession * session, char * hosts, char * message)
LONG QRequestPrivilegesOp (struct QSession * session, ULONG privBits)
```

LONG **QReleasePrivilegesOp** (struct QSession * session, ULONG privBits)
LONG **QKillClientsOp** (struct QSession * session, char * hosts)
LONG **QSetParameterOp** (struct QSession * session, char * paramName, char * newVal)
LONG **QGetParameterOp** (struct QSession * session, char * paramName)
BOOL **QDetachSession** (struct QSession * session, ULONG flags)
BOOL **QReattachSession** (struct QSession * session, ULONG flags)
LONG **QGo** (struct QSession * session, ULONG flags)
void **FreeQMessage** (struct QSession * session, struct QMessage * qmsg)
ULONG **QNumQueuedPackets** (struct QSession * session);
ULONG **QNumQueuedBytes** (struct QSession * session);
char * **QErrorMessage** (struct QSession * session, LONG statusCode);
Library functions for ARexx only:
GetQMessageField (message, fieldname)
GetNextQMessage (session, timeout)

1.18 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.

1.19 qmessages

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

In C or Pascal, These QMessages can be accessed by your program by waiting on the qMsgPort field in the QSession struct that is returned by [QNewSession](#) . In ARexx, you can receive QMessages

by calling `GetNextQMessage` .

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.

Note that ARexx scripts cannot read these fields directly; instead, they should use the `GetQMessageField` function to access them.

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.

QERROR_SYS_MESSAGE

This message is a system message `sent` to you by a client with `QPRIV_SENDSYSMESSAGES` access. The `qm_Path` field will tell you who sent it to you, and the `qm_Data` field is a NULL-terminated ASCII string that contains the text of the message.

Note that you will only ever see this value of `qm_Status` if you have `requested` `QPRIV_GETSYSMESSAGES` access.

QERROR_SEND_DONE

This message was sent to you by the AMarquee TCP thread running on your machine, to let you know that your `QGo()` packet has been sent (or at least, that it has been completely handed off to the TCP Stack's transmission code). You will only get this `QMessage` back if you specified the `QGOF_NOTIFY` flag in your call to `QGo()`.

The `qm_ID` field of this `QMessage` is the same as the return value of the `QGo()` call that caused it to be sent. (This status type is new for v47 of `amarquee.library`)

- 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 is usually equal to qm_DataLen, but it 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 AMarquee 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!

1.20 qnewsession

amarquee.library/QNewSession amarquee.library/QNewSession

NAME

QNewSession - Creates a new connection to an AMarquee 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 AMarquee 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" AMarquee 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("example.server.com", 2957, "ExampleProgram"))
printf("Connection to example.server.com:2957 was successful\n");
else
printf("Connection failed.\n");
```

AREXX NOTES

For ARExx, the return value of this method is a string representation of the pointer to the session struct. If the returned value is equal to zero, then the call failed. Otherwise, use the return value as a handle to pass to other amarquee.library functions.

Note that instead of calling Wait() and GetMsg(), you will want to use amarquee.library's `GetNextQMessage` function to receive incoming QMessages.

AREXX EXAMPLE

```
session = QNewSession('example.server.com', 2957, 'ExampleScript')
if (session > 0) then say "Connection was successful!"
else say "Connection failed!"
```

SEE ALSO

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

1.21 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.

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 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("example.server.com", 2957, "ExampleProgram"))
{
printf("Connecting to example.server.com: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");

```

AREXX NOTES

For ARexx, the return value of this method is a string representation of the pointer to the session struct. If the returned value is equal to zero, then the call failed. Otherwise, use the return value as a handle to pass to other amarquee.library functions.

Note that instead of calling Wait() and GetMsg(), you will want to use amarquee.library's [GetNextQMessage](#) function to receive incoming QMessages.

AREXX EXAMPLE

```

session = QNewSessionAsync('example.server.com', 2957, 'ExampleScript')
if (session > 0) then say "Call was successful."
else say "Call failed!"
/* Now wait for QMessages with GetNextQMessage... */

```

SEE ALSO

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

1.22 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/programe"`). 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.

`programes` - 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");
```

AREXX NOTES

For ARExx, the return value of this method is a string representing two numbers: A pointer to the session struct, and the port number that is being used. The return value of a successful call might look like this:

```
"1254115 1024"
```

If the first number is zero, then the call failed.

Otherwise, use the return value as a handle to pass to other amarquee.library functions. You may read the port (e.g. second) number and use it as you see fit (e.g. send it to another client so they know what port they can connect to you at). You do not need to remove the port number from the string before passing the string as a handle to subsequent amarquee.library calls--they will ignore the extra number.

Note that instead of calling Wait() and GetMsg(), you will want to use amarquee.library's [GetNextQMessage](#) function to receive incoming QMessages.

AREXX EXAMPLE

```
session = QNewHostSession('example.server.com', 0, 'ExampleScript')
if (session > 0) then say "Connection was successful!"
else say "Connection failed!"
```

SEE ALSO

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

1.23 qnewserveression

amarquee.library/QNewServerSession amarquee.library/QNewServerSession

NAME

QNewServerSession - Link to the socket provided to your program

by inetd when you have been launched as

a server daemon by your TCP stack's inetd.

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 NOT work with the Inet225 version of amarqueed.library. This is because under Inet225 the inherited socket is passed via main's() argc and argv parameters, which this call does not have access to.

Email [me](#) if you need this call to work under Inet225; if there is demand I will try and work out a solution.

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](#) 'd 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");
```

AREXX NOTES

For ARExx, the return value of this method is a string representation of the pointer to the session struct. If the returned value is equal to zero, then the call failed. Otherwise, use the return value as a handle to pass to other amarquee.library functions.

Note that instead of calling Wait() and GetMsg(), you will want to use amarquee.library's [GetNextQMessage](#) function to receive incoming QMessages.

Note that in order for this command to be useful, your ARExx script must have been launched by inetd. I'm not sure how that is done...

AREXX EXAMPLE

```
session = QNewServerSession('#?', '#?')
if (session > 0) then say "Call was successful!"
else say "Call failed!"
```

SEE ALSO

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

1.24 qfreesession

amarquee.library/QFreeSession amarquee.library/QFreeSession

NAME

QFreeSession - Closes the given connection to an AMarquee server.

SYNOPSIS

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

```
LONG 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...

INPUTS

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

RESULTS

QFreeSession always succeeds. However, QFreeSession returns a QERROR_* code describing any earlier problems--usually the return code is QERROR_NO_ERROR, but if there was a problem in connecting this QSession, or an exceptional event (such as the TCP stack shutting down) occurred, then the return value might be one of the following:

QERROR_NO_CLIENT_MEM - The client computer ran out of memory.

QERROR_NO_TCP_STACK - Couldn't connect because the TCP stack was not running.

QERROR_HOSTNAME_LOOKUP - The name server failed to find the requested host.

QERROR_ABORTED - The TCP thread received a CTRL-C signal.

This happens when the TCP stack is shutting down, or if the user is monkeying around with system monitors. ;)

QERROR_NO_SERVER - The requested host refused the TCP connection, most likely because an AMarquee server was not installed properly on that machine.

QERROR_NO_INETD - [QNewServerSession](#) detected that InetD was

not what invoked the user program.

QERROR_ACCESS_DENIED - The AMarquee server program refused to accept the connection.

This return value is for informational purposes only; no specific actions are required based on the value returned.

EXAMPLE

```
struct QSession * s;
if (s = QNewSession("example.server.com", 2957, "ExampleProgram"))
{
LONG err;
/* ... do stuff with the session here ... */
/* ... */
err = QFreeSession(s);
if (ret != QERROR_NO_ERROR)
printf("Error connecting session: %s\n",QErrorName(err));
}
```

AREXX NOTES

Note that it is *critical* that your script call this function for each active QSession, before the script terminates! Otherwise, the connection threads will hang around and probably crash the computer. The best way to make sure that all QSessions are freed is to use ARExx's "signal on" exception handling to clean up.

AREXX EXAMPLE

```
signal on break_c
session = QNewSession('example.server.com', 2957, 'exampleRexx')
if (session > 0) then do
/* ... */
break_c:
call QFreeSession(session)
end
```

SEE ALSO

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

1.25 qdetachsession

amarquee.library/QDetachSession amarquee.library/QDetachSession

NAME

QDetachSession - Removes the messaging connection between the given QSession and the user process that created it.

SYNOPSIS

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

```
BOOL QDetachSession(struct QSession * session, ULONG flags)
```

FUNCTION

When a QSession is created, it is implicitly coupled with (a.k.a. "attached to") the process that created it. That is to say, only the process that created the QSession will be able to successfully receive QMessages from that QSession. However, there are some cases where it would be convenient to "hand off" a QSession from one process to another. In order to do this, you must first call QDetachSession on the session in the process that created it, then call **QReattachSession** on the session in the process you wish to hand the QSession to.

NOTE

This function requires v47 of amarquee.library.

This function is only useful if you want to pass QSessions from one process to another. Most AMarquee programs won't need to use it.

Once you have detached a QSession, there are special rules you must follow. These rules are:

- Only the process that is attached to a QSession may call QDetachSession() on it.
 - The qMsgPort field of a detached QSession is NULL. As such, you may not Wait() or GetMsg() or otherwise access the QSession's qMsgPort field in any way (until you reattach the QSession, anyway). When the QSession is reattached, the qMsgPort field will have a new value (so don't store the old value, as it becomes invalid when QDetachSession() is called)
 - An unattached QSession may be **freed** by any process. Attached QSessions may only be freed by the process they are attached to. Attempting to free a QSession that is attached to another process will most likely hang or crash your process.
 - All AMarquee functions (except **GetNextQMessage** in ARexx) may be called on a detached QSession, but you won't be able to get results (i.e. QMessages) back from it until the QSession has been reattached.
 - All QMessages that the detached QSession generates will
-

be stored in an internal queue, and will be made available when the QSession is reattached. That is to say, events will not be "dropped" just because a QSession is detached. (which means if you leave a QSession detached for a long time, it may build up quite a large queue of stored QMessages and eat up lots of memory)

- It's (still) not safe to let multiple processes accessing a QSession simultaneously. That is, the Q*Op() methods should be serialized, and the Wait()'ing and GetMsg()'ing should only be done by the one thread that the QSession is attached to.

Finally, detaching or **reattaching** a QSession to a process is a synchronous operation, so if the TCP thread is busy shovelling lots of data around, it may take a while for this function to return.

INPUTS

session - Pointer to the QSession struct you wish to detach.

flags - Reserved for future use. Always set to 0L for now.

RESULTS

Returns TRUE on success, FALSE on failure. (Failure occurs if there is not enough memory, or if the QSession was already detached)

EXAMPLE

See the included program AMarqueeDebugMultiThread.c for a full working example of how to use this function.

```
struct QSession * s;
/* ... In thread one ... */
if (s = QNewSession("example.server.com", 2957, "ExampleProgram"))
{
/* ... do stuff with the session here ... */
if (QDetachSession(s, 0L))
{
GiveSessionToAnotherThread(s);
}
else printf("Uh oh, QDetachSession failed!\n");
}
-----
/* ... In thread two ... */
s = GetSessionFromFirstThread(); /* using message passing or whatever */
```

```

if (QReattachSession(s, 0L))
{
/* ... Do stuff with the QSession ... */
QFreeSession(s);
}
else printf("Oh dear, QReattachSession failed!\n");

```

AREXX NOTES

QDetachSession() is implemented for ARexx, although I have yet to see a multithreaded ARexx script. Perhaps it might come in handy for passing a QSession from one ARexx script to another, though.

The ARexx function returns 1 on success, 0 on failure.

```

if (QDetachSession(session) == 0) then
say "uh oh, session detach failed!"
end

```

SEE ALSO

[QReattachSession](#) , [QFreeSession](#)

1.26 greattachsession

amarquee.library/QReattachSession amarquee.library/QReattachSession

NAME

QReattachSession - Creates a messaging connection between the given detached QSession and calling process.

SYNOPSIS

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

```
BOOL QReattachSession(struct QSession * session, ULONG flags)
```

FUNCTION

Once a QSession has been **detached** from its process, this function may be called by any process in order to create a messaging association between the calling process and the QSession.

After this function returns (successfully), the QSession is no longer detached, and is operable by the calling process, just as if the calling process had created the QSession itself. In particular, the QSession's qMsgPort field will no longer be NULL: rather, it will be set to point to a MsgPort that the calling process can then Wait() on for QMessages.

NOTE

This function requires v47 of amarquee.library.

When a QSession is first created, it is already attached to its creating process. Thus, this function is only useful if you want to pass QSessions from one process to another.

Most AMarquee programs won't need to use it.

See [QDetachSession](#) for the list extra rules that apply to detached QSession's.

Reattaching a QSession to a process is a synchronous operation, so if the TCP thread is busy shovelling lots of data around, it may take a while for this function to return.

INPUTS

session - Pointer to the QSession struct you wish to reattach.

flags - Reserved for future use. Always set to 0L for now.

RESULTS

Returns TRUE on success, FALSE on failure. (Failure occurs if there is not enough memory, or if the QSession was already attached to a process)

EXAMPLE

See the included program AMarqueeDebugMultiThread.c for a full working example of how to use this function.

```
struct QSession * s;
/* ... In thread one ... */
if (s = QNewSession("example.server.com", 2957, "ExampleProgram"))
{
/* ... do stuff with the session here ... */
if (QReattachSession(s, 0L))
{
GiveSessionToAnotherThread(s);
}
else printf("Uh oh, QReattachSession failed!\n");
}
-----
/* ... In thread two ... */
s = GetSessionFromFirstThread(); /* using message passing or whatever */
if (QReattachSession(s, 0L))
{
/* ... Do stuff with the QSession ... */
QFreeSession(s);
}
```

```
else printf("Oh dear, QReattachSession failed!\n");
```

AREXX NOTES

QReattachSession() is implemented for ARexx, although I have yet to see a multithreaded ARexx script. Perhaps it might come in handy for passing a QSession from one ARexx script to another, though.

The ARexx function returns 1 on success, 0 on failure.

```
if (QReattachSession(session) == 0) then
```

```
say "uh oh, session reattach failed!"
```

```
end
```

SEE ALSO

[QDetachSession](#) , [QFreeSession](#)

1.27 qnumqueuedpackets

amarquee.library/QNumQueuedPackets amarquee.library/QNumQueuedPackets

NAME

QNumQueuedPackets - Returns (approximately) the number of messages awaiting processing by the QSession's TCP thread.

SYNOPSIS

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

```
ULONG QNumQueuedPackets(struct QSession * session)
```

FUNCTION

Returns the number of messages that have been sent to the TCP thread for this QSession, but have not yet been processed by the TCP thread. This value includes one point for each transaction generated by the Q*Op() calls, as well as one point per QGo() call. No thread synchronization is done, so the value returned should be considered approximate only. This function can be used to see if packets are being queued faster than they are being sent, and thus take corrective action before too much memory is allocated for queued packets.

INPUTS

session - Pointer to the QSession struct you want information for.

RESULTS

Number of queued messages.

EXAMPLE

```
struct QSession * s;
```

```
if (s = QNewSession("example.server.com", 2957, "ExampleProgram"))
```

```

{
ULONG numQueued;
/* ... */
numQueued = QNumQueuedPackets(s);
printf("Currently there are %i packets awaiting transmission\n",numQueued);
/* ... */
QFreeSession(s);
}

```

SEE ALSO

[QNumQueuedBytes](#)

1.28 qerrorname

amarquee.library/QErrorName amarquee.library/QErrorName

NAME

QErrorName - Returns an ASCII string describing a QERROR_* code

SYNOPSIS

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

```
char * QErrorName(LONG errno)
```

FUNCTION

Returns a pointer to a string that describes the QERROR_* code passed in as an argument.

INPUTS

errno - a QERROR_* error code.

RESULTS

Pointer to a string describing the error. This string is owned by amarquee.library and should be considered read-only. It will be valid until amarquee.library is closed.

EXAMPLE

```
printf("Error QERROR_NO_CLIENT_MEM means %s\n",
QErrorName(QERROR_NO_CLIENT_MEM));
```

AREXX EXAMPLE

```
say "Error QERROR_NO_CLIENT_MEM means: " || QErrorName(QERROR_NO_CLIENT_MEM)
```

1.29 qnumqueuedbytes

amarquee.library/QNumQueuedBytes amarquee.library/QNumQueuedBytes

NAME

QNumQueuedBytes - Returns (approximately) the number of bytes of

user data are awaiting transmission by the QSession's TCP thread.

SYNOPSIS

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

```
ULONG QNumQueuedBytes(struct QSession * session)
```

FUNCTION

Returns the number of bytes of data that have been sent to the TCP thread for this QSession, but have not yet been sent to the TCP stack by the TCP thread. This value does not include any "overhead" introduced by the AMarquee system messages or headers--it only tracks the sum of sizes of the user's queued data buffers.

No thread synchronization is done, so the value returned should be considered approximate only. This function can be used to see if bytes are being queued faster than they are being sent, and thus take corrective action before too much memory is allocated for queued packets.

INPUTS

session - Pointer to the QSession struct you want information for.

RESULTS

Number of bytes currently being used to store queued transactions.

EXAMPLE

```
struct QSession * s;
if (s = QNewSession("example.server.com", 2957, "ExampleProgram"))
{
    ULONG numQueued;
    /* ... */
    numQueued = QNumQueuedBytes(s);
    printf("Currently there are %i bytes of data awaiting transmission\n",numQueued);
    /* ... */
    QFreeSession(s);
}
```

SEE ALSO

[QNumQueuedPackets](#)

1.30 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");
```

AREXX EXAMPLE

```
transID = QDebugOp(session, 'An ARexx debug string')
```

```
if (transID = 0) then say "Oops, operation failed"
```

SEE ALSO

[QGo](#)

1.31 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");
```

AREXX NOTES

In ARExx, the third argument is optional. If it is not specified, -1 (no size limit) will be assumed.

AREXX EXAMPLE

```
transID = QGetOp(session, '/#?/#?/testNode')
```

```
if (transID = 0) then say "Oops, operation failed"
```

SEE ALSO

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

1.32 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");
```

AREXX EXAMPLE

```
transID = QDeleteOp(session, 'myNode')
```

```
if (transID = 0) then say "Oops, operation failed"
```

SEE ALSO

[QGo](#)

1.33 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");
```

AREXX EXAMPLE

```
transID = QRenameOp(session, 'myDir/node1', 'node2')
if (transID = 0) then say "Oops, operation failed"
```

SEE ALSO

[QGo](#)

1.34 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 **QGetAndSubscribeOp** instead.

If a client whose data you are watching disconnects, you will get messages indicating that each node you are watching has been deleted. (If you want to know for sure that a disconnection is taking place and not just a bunch of manual deletions, watch the client's root node (e.g. `"/progName/clientName"`)--this node will only ever be deleted when the client disconnects.

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 AMarqueued 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");
```

AREXX NOTES

In ARExx, the third argument is optional. If it is not specified, it defaults to -1 (e.g. no size limit).

AREXX EXAMPLE

```
transID = QSubscribeOp(session, '#?/ExampleProgram/data')
if (transID = 0) then say "Oops, operation failed"
```

SEE ALSO

[QGo](#) , [QGetAndSubscribeOp](#) , [QClearSubscriptionOp](#)

1.35 qgetandsubscribeop

amarquee.library/QGetAndSubscribeOp amarquee.library/QGetAndSubscribeOp

NAME

QGetAndSubscribeOp - Get the given data items, and start watching them.

SYNOPSIS

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

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

NOTE

This function required v46 of amarquee.library, and v46 of AMarqueued.

FUNCTION

This function is a combination of the [QGetOp](#) () and [QSubscribeOp](#) () methods. Calling this function is the same as calling [QGetOp](#) () followed by [QSubscribeOp](#) (), with the same arguments. This function is handy when you need to know the current state of some data and also want to be notified when the data changes.

The advantages of this function over calling [QGetOp](#) () and

QSubscribeOp () separately are twofold: first off, you only have to remember the ID number of one operation, rather than two; and secondly, it eliminates the possibility of race conditions between the two operations.

INPUTS

session - The session to send the get&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 get&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 the current value of the item
"data" in "ExampleProgram"'s home directory, and also to
send us a message whenever ExampleProgram changes the value
of "data". */
```

```
if (transID = QGetAndSubscribeOp(session, "/#?/ExampleProgram/data", -1))
```

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

```
else
```

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

AREXX NOTES

In ARExx, the third argument is optional. If it is not specified, it defaults to -1 (e.g. no size limit).

AREXX EXAMPLE

```
transID = QGetAndSubscribeOp(session, '/#?/ExampleProgram/data')
```

```
if (transID = 0) then say "Oops, operation failed"
```

SEE ALSO

QGo , **QGetOp** , **QSubscribeOp** , **QClearSubscriptionOp**

1.36 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");
```

AREXX NOTES

In ARexx, the fourth argument is optional. If it is not specified, then the length of the third argument will be computed automatically so that the entire ARexx string is sent. (that is, it defaults to `length(data)+1`. Note that if you specify the fourth argument to be a value of less than that, the first (n) bytes of the string will be stored on the server as an unterminated character array)

AREXX EXAMPLE

```
transID = QSetOp(session, 'myName', 'Fred')
if (transID = 0) then say "Oops, operation failed"
```

SEE ALSO

[QGo](#) , [QStreamOp](#)

1.37 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");
```

AREXX NOTES

In ARexx, the fourth argument is optional. If it is not specified, then the length of the third argument will be computed automatically so that the entire ARexx string is sent. (that is, it defaults to `length(data)+1`. Note that if you specify the fourth argument to be a value of less than that, it will be stored on the server as an unterminated character array!)

AREXX EXAMPLE

```
transID = QMessageOp(session, '/#?/ExampleProgram', 'Hey Everybody!')
if (transID = 0) then say "Oops, operation failed"
```

SEE ALSO

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

1.38 qsysmessageop

amarquee.library/QSysMessageOp amarquee.library/QSysMessageOp

NAME

QSysMessageOp - Send a system message to one or more other clients.

SYNOPSIS

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

```
LONG QSysMessageOp(struct QSession * session, char * hosts, char * message)
```

FUNCTION

This function allows you to send a text message to all clients who have **requested** `QPRIV_GETSYSMESSAGE` access. Note that in order to use this function, your client must have `QPRIV_SENDSYSMESSAGE` access.

Your message will be sent to every host that matches the regular expression specified in the "hosts" argument, as long as the client has `QPRIV_GETSYSMESSAGE` access.

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 do not have `QPRIV_GETSYSMESSAGE` access), you will receive a `QERROR_UNPRIVILEGED` `QMessage` notifying you of this. You will also receive a

QERROR_UNPRIVILEGED QMessage if you did not have QPRIV_SENDSYSMESSAGE access when you called this function.

When another client sends your client a QSysMessageOp, 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").

The qm_Status field will be set to QERROR_SYS_MESSAGE.

The qm_Data field will contain the ASCII text of the message body.

NOTE

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

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".

message - A pointer to the text string you wish to transmit.

If NULL is specified, a one-byte empty string ("") will be sent.

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

```
/* Send a system message to all clients who are listening for it. */
if (transID = QSysMessageOp(session, "/#?/#?", "System shutting down! Log off now.");
printf("SysMessage op succeeded, was given id #li\n",transID);
else
printf("SysMessage op failed. (no memory?)\n");
```

AREXX EXAMPLE

```
transID = QSysMessageOp(session, '/#?/#?', 'System shutting down!')
if (transID = 0) then say "Oops, operation failed"
```

SEE ALSO

[QGo](#) , [QRequestPrivilegesOp](#) , [QMessageOp](#)

1.39 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 AMarqueued.

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");

```

AREXX NOTES

In ARexx, the fourth argument is optional. If it is not specified, then the length of the third argument will be computed automatically so that the entire ARexx string is sent. (that is, it defaults to `length(data)+1`. Note that if you specify the fourth argument to be a value of less than that, it will be stored on the server as an unterminated character array!)

AREXX EXAMPLE

```

transID = QStreamOp(session, 'myName', 'Fred')
if (transID = 0) then say "Oops, operation failed"

```

SEE ALSO

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

1.40 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 = QClearSubscriptionsOp(session, 0))
printf("ClearSub op succeeded, was given id #%li\n",transID);
else
printf("ClearSub op failed. (no memory?)\n");
```

AREXX NOTES

In ARExx, the second argument is optional. If not specified, it defaults to zero (e.g. clear all subscriptions)

AREXX EXAMPLE

```
transID = QClearSubscriptionsOp(session)
if (transID = 0) then say "Oops, transaction failed."
```

SEE ALSO

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

1.41 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** AMarqueed 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");
```

AREXX EXAMPLE

```
transID = QPingOp(session)
if (transID = 0) then say "Oops, transaction failed."
```

SEE ALSO

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

1.42 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_Alloced; /* The number of bytes currently allocated by your server. */
LONG qr_Avail; /* The number of bytes that may actually be allocated by your server. */
ULONG qr_CurrentPrivs; /* Bit chord of QPRIV_* values: privileges you currently enjoy */
ULONG qr_PossiblePrivs; /* Bit chord of QPRIV_* values: privileges you can acquire */
};
```

qr_Allowed will be constant throughout your AMarquee session.

qr_Alloced 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.

qr_CurrentPrivs can be set via calls to [QRequestPrivilegesOp](#) or [QReleasePrivilegesOp](#). It can never contain bits that are not also in qr_PossiblePrivs field, though.

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");
```

AREXX EXAMPLE

```
transID = QInfoOp(session)
```

```
if (transID = 0) then say "Oops, transaction failed."
```

SEE ALSO

[QGo](#) , [QPingOp](#)

1.43 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 see 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");
```

AREXX EXAMPLE

```
transID = QSetAccessOp(session, '#?/ExampleProgram')
if (transID = 0) then say "Oops, transaction failed."
```

SEE ALSO

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

1.44 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");
```

AREXX EXAMPLE

```
transID = QSetMessageAccessOp(session, '#?/ExampleProgram')
if (transID = 0) then say "Oops, transaction failed."
```

SEE ALSO

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

1.45 qrequestprivilegesop

amarquee.library/QRequestPrivilegesOp amarquee.library/QRequestPrivilegesOp

NAME

QRequestPrivilegesOp - Ask for special privileges/abilities.

SYNOPSIS

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

```
LONG QRequestPrivilegesOp(struct QSession * session, ULONG privBits)
```

FUNCTION

This function lets your client ask the server for permission to do certain operations that are otherwise restricted.

The server may or may not actually grant you permission.

NOTE

This function requires v45 of amarquee.library, and v1.45 of AMarqueued.

INPUTS

session - The session for which you wish to request privileges.

privBits - A bit-chord specifying which special privileges you want.

The following constants are currently available:

QPRIV_KILLCLIENTS - The ability to use [QKillClientsOp](#) to disconnect other client programs.

QPRIV_SENDSYSMESSAGES - The ability to [send system messages](#) to other client programs.

QPRIV_GETSYSMESSAGES - The ability to receive system messages sent by other programs. (This privilege is guaranteed to be granted to any client that requests it)

QPRIV_ADMIN - The ability to change system settings via [QSetParameterOp](#) .

QPRIV_ALLPRIVILEGES - This constant is equivalent to all the above constants OR'd together.

RESULTS

Returns the assigned ID number of the access operation, or 0 to indicate failure (due to memory shortage).

To find out if your requests were granted, call [QInfoOp](#) afterwards and read the qr_Privs field of the returned struct to find your new permissions status.

EXAMPLE

LONG transID;

/* Request the ability to get system messages and to disconnect other clients */

```
if (transID = QRequestPrivilegesOp(session, QPRIV_KILLCLIENTS | QPRIV_GETSYSMESSAGES))
```

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

```
else
```

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

AREXX EXAMPLE

```
transID = QRequestPrivilegesOp(session, QPRIV_KILLCLIENTS | QPRIV_GETSYSMESSAGES)
```

```
if (transID = 0) then say "Oops, transaction failed."
```

SEE ALSO

[QGo](#) , [QReleasePrivilegesOp](#) , [QKillClientsOp](#)

1.46 qreleaseprivilegesop

amarquee.library/QReleasePrivilegesOp amarquee.library/QReleasePrivilegesOp

NAME

QReleasePrivilegesOp - Relinquish special privileges/abilities.

SYNOPSIS

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

```
LONG QReleasePrivilegesOp(struct QSession * session, ULONG privBits)
```

FUNCTION

With this function you can give up your special permissions that were previously given you as the result of a call to [QRequestPrivilegesOp](#).

NOTE

This function requires v45 of amarquee.library, and v1.45 of AMarqueed.

INPUTS

session - The session for which you wish to release privileges.
 privBits - A bit-chord specifying which special privileges you want to give up. The following constants are currently available:
 To see which constants may be or'd together into privBits, [read the QRequestPrivilegesOp documentation](#).

RESULTS

Returns the assigned ID number of the access operation, or 0 to indicate failure (due to memory shortage).
 The server will always allow you to give up privileges, so no further result checking should be necessary.

EXAMPLE

```
LONG transID;
/* Release the ability to kill other clients */
if (transID = QReleasePrivilegesOp(session, QPRIV_KILLCLIENTS))
printf("ReleasePrivileges op succeeded, was given id #%li\n",transID);
else
printf("ReleasePrivileges op failed. (no memory?)\n");
```

AREXX EXAMPLE

```
transID = QReleasePrivilegesOp(session, QPRIV_KILLCLIENTS)
if (transID = 0) then say "Oops, transaction failed."
```

SEE ALSO

[QGo](#), [QRequestPrivilegesOp](#), [QKillClientsOp](#)

1.47 qkillclientsop

amarquee.library/QKillClientsOp amarquee.library/QKillClientsOp

NAME

QKillClientsOp - Cause other clients to be disconnected from the server.

SYNOPSIS

```
#include <clib/amarquee_protos.h>
LONG QKillClientsOp(struct QSession * session, char * hosts)
```

FUNCTION

With this function you can forcibly remove the AMarqueued connections from other client programs that are logged in to the same server.

This function is intended to be used by administration programs only, and thus it requires special permission to use.

NOTE

This function requires v45 of amarquee.library, and v1.45 of AMarqueued.

This command will never cause your own client to be disconnected, even if your /hostName/progName is included in the hosts argument's specification.

INPUTS

session - The session on which you wish to kill other clients.

hosts - A hostname specifier describing which hosts and/or programs should be disconnected. This should be a string in the form of "/hostExp/progExp".

RESULTS

Returns the assigned ID number of the access operation, or 0 to indicate failure (due to memory shortage, or because the hosts string was not formatted correctly).

If you do not have QPRIV_KILLCLIENTS permission, this command will have no effect other than to have a QERROR_UNPRIVILEGED packet be sent to you.

EXAMPLE

LONG transID;

```
/* Disconnect all clients except my own */
```

```
if (transID = QKillClientsOp(session, "/#?/#?"))
```

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

```
else
```

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

AREXX EXAMPLE

```
transID = QKillClientsOp(session, '/#?/#?')
```

```
if (transID = 0) then say "Oops, transaction failed."
```

SEE ALSO

[QGo](#) , [QRequestPrivilegesOp](#) , [QReleasePrivilegesOp](#)

1.48 getnextqmessage

amarquee.library/GetNextQMessage amarquee.library/GetNextQMessage

NAME

GetNextQMessage - Wait for next QMessage to be received from the TCP thread.

SYNOPSIS

GetNextQMessage(session, timeout, signals)

FUNCTION

This function is available from ARexx scripts only.

It replaces most of the functionality of the C Wait() call, and allows your ARexx script to sleep until a QMessage is ready, or an (optional) timeout occurs, or an (optional) signal is raised.

INPUTS

session - The session pointer string that was returned by **QNewSession** (or a related function) to create this session.

timeout - The number of milliseconds to wait before timing out and returning. Set to zero to just poll for QMessages.

If this argument is set to less than zero, or is not specified, then no timeout will occur (i.e. the function will not return until a QMessage is available, or a signal is raised).

signals - A string specifying which interrupt signals should cause this function to return immediately. If not specified, any interrupt signal will cause this function to return.

(That is, this argument defaults to the value:

```
'SIGBREAKF_CTRL_C|SIGBREAKF_CTRL_D|SIGBREAKF_CTRL_E|SIGBREAKF_CTRL_F')
```

RESULTS

Returns a pointer string to the QMessage if one became available, or zero if it had to return for some other reason (a signal was raised or the timeout period expired). Note that signals CTRL-C, CTRL-D, CTRL-E, and CTRL-F will all cause this function to return, and also the signals will be propagated to the ARexx script (so it should be prepared to handle them!)

NOTE

It is an error to call GetNextQMessage() on a **detached** QSession.

AREXX EXAMPLE

```
/* Will return a QMessage is received, or when 5 seconds elapses,
```

```

or any CTRL-(C,D,E,F) combo is pressed. */
message = GetNextQMessage(session, 5000)
if (message = 0) then say "Oops, timed out or signalled!"
else do
say "Got QMessage with id " || GetMessageField(message, 'ID')
/* ... */
call FreeQMessage(session, message) /* must do this, or memory leaks */
end
AREXX EXAMPLE TWO
/* Will return when a QMessage is received, or when CTRL-C or
CTRL-D are pressed */
message = GetNextQMessage(session, -1, 'SIGBREAKF_CTRL_C|SIGBREAKF_CTRL_D')
if (message = 0) then say "Oops, timed out or signalled!"
else do
say "Got QMessage with id " || GetMessageField(message, 'ID')
/* ... */
call FreeQMessage(session, message) /* must do this, or memory leaks */
end
SEE ALSO

```

[GetQMessageField](#)

1.49 getqmessagefield

amarquee.library/GetQMessageField amarquee.library/GetQMessageField

NAME

GetQMessageField - Returns one of the fields of a QMessage.

SYNOPSIS

GetQMessageField(message, fieldname)

FUNCTION

This function is available from ARexx scripts only.

It allows you to get at the various fields of the QMessage struct returned to you by [GetNextQMessage](#), without having to muck around with byte offsets and such.

INPUTS

message - A non-zero pointer string returned by [GetNextQMessage](#).

fieldname - A string specifying which field you wish to have returned. Should be one of the following, and is case-insensitive:

ID : Message ID # of transaction related to this opCode, or -1 if no ID is applicable.

Status : One of the **QERROR_* identifiers** .

Path : Pathname of a node, or NULL if not applicable.

Data : Data buffer, or "" if not applicable.

DataLen : Length of qm_Data buffer, or 0 if not applicable.

ActualLen : Length of the data buffer stored on the AMarquee server.

ErrorLine : Line # of the server source code that generated the error. Useful for debugging.

(The prefix "qm_" may be added to any of these keywords, if you prefer to have the fields exactly match those of the QMessage struct defined in amarquee.h)

For QMessages that contain a QRunInfo struct for their qm_Data (i.e. those returned in response to a QInfoOp), you may also specify the following fields (with or without a "qr_" prefix):

Allowed : The theoretical maximum number of bytes your server may allocate.

Alloced : The number of bytes currently allocated by your server.

Avail : The number of bytes that may actually be allocated by your server.

CurrentPrivs : 'l' separated list of QPRIV_* identifiers, showing what special privs you have.

PossiblePrivs : 'l' separated list of QPRIV_* identifiers, showing what special privs you could get if you asked.

RESULTS

Returns a string containing the contents of the requested part of the QMessage.

AREXX EXAMPLE

```
message = GetNextQMessage(session)
if (message > 0) then do
say "Got Message, path was " || GetQMessageField(message, 'Path')
call FreeQMessage(session, message)
end
```

SEE ALSO

[GetNextQMessage](#)

1.50 paramprivs

Below is a table showing the parameter name strings recognized by **QSetParameterOp** and **QGetParameterOp** , and what privileges you need to have in order to get or set each of them.

Many of the entries (all beginning with AMARQUEED_, in fact) correspond to **environment variables** on the server computer. Calling **GetParameterOp** and **SetParameterOp** with these parameter names will have the effect of getting and setting the like-named environment variable on the server computer. Note that AMarqueued daemons only read environment variables when

a new connection is first established, so changing these values will not affect current connections.

Non-environmental-variable parameters are described separately below.

```

+-----+
| Sets ENV Var Privs needed Privs needed |
| Parameter name on server? to Get value to Set value |
+-----+
||
| AMARQUEED_MAXQUEUEDMESSAGES YES none QPRIV_ADMIN |
||
| AMARQUEED_BANNED1 YES QPRIV_ADMIN QPRIV_ADMIN |
||
| AMARQUEED_MAXCONN YES none QPRIV_ADMIN |
||
| AMARQUEED_TOTALMAXCONN YES QPRIV_ADMIN QPRIV_ADMIN |
||
| AMARQUEED_KILLCLIENTS YES QPRIV_ADMIN QPRIV_ADMIN |
||
| AMARQUEED_SENDSYSMESSAGES YES QPRIV_ADMIN QPRIV_ADMIN |
||
| AMARQUEED_ALLPRIVILEGES YES QPRIV_ADMIN (unsettable) |
||
| AMARQUEED_ADMIN YES QPRIV_ADMIN (unsettable) |
||
| AMARQUEED_DEBUG YES none QPRIV_ADMIN |
||
| AMARQUEED_PRIORITY YES none QPRIV_ADMIN |
||
| AMARQUEED_MAXMEM YES none QPRIV_ADMIN |
||
| AMARQUEED_MINFREE YES QPRIV_ADMIN QPRIV_ADMIN |
||
| AMARQUEED_PINGRATE YES none QPRIV_ADMIN |
||
| SERVERVERSION NO none (unsettable) |
||
| MAXQUEUEDMESSAGES NO none none2 |
+-----+

```

(¹) Includes AMARQUEED_BANNED, AMARQUEED_BANNED1, AMARQUEED_BANNED2, all the way up to AMARQUEED_BANNED9

(²) no privileges needed to lower value; QPRIV_ADMIN needed to raise it.

Miscellaneous parameter descriptions

- SERVERVERSION: This read-only parameter returns the full version string of the server daemon you are connected to.

- MAXQUEUEDMESSAGES: This parameter lets you get or set the maximum amount of inter-process messages your server daemon can accumulate before it is killed. The value is originally set to the value specified by the AMARQUEED_MAXQUEUEDMESSAGES parameter on the server, and may be changed by each client after that.

Any client may lower the value; QPRIV_ADMIN is needed to raise it. Values of 0 or less are interpreted to mean "a very large limit".

1.51 qsetParameterop

amarquee.library/QSetParameterOp amarquee.library/QSetParameterOp

NAME

QSetParameterOp - Set miscellaneous parameters.

SYNOPSIS

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

```
LONG QSetParameterOp(struct QSession * session, char * paramName, char * newValue)
```

FUNCTION

This function lets your client set certain environment variables or other miscellaneous parameters on the server machine. Note that your client will need QPRIV_ADMIN status to set most of these variables.

NOTE

This function requires v45 of amarquee.library, and v1.45 of AMarqueed.

INPUTS

session - The session for which you wish to set a parameter.

paramName - The keyword corresponding to the parameter you wish to set. Keywords, and who they may be set by, are described in this [table](#).

Note that parameter names are case sensitive!

newValue - A NUL-terminated string that is the value you wish to set the given parameter to.

Values are always passed in ASCII format, even for numeric parameters.

RESULTS

Returns the assigned ID number of the access operation, or 0 to indicate failure (due to memory shortage).

If you did not have permission to set the parameter you indicated, a `QERROR_UNPRIVILEGED` message will be sent back from the server.

If you specified an unknown paramName, a `QERROR_MALFORMEDKEY` will be sent back from the server.

EXAMPLE

LONG transID;

```
/* Try to make the server execute a "setenv AMARQUEED_PINGRATE 12"... */
if (transID = QSetParameterOp(session, "AMARQUEED_PINGRATE", "12"))
printf("Set parameter op succeeded, was given id #li\n",transID);
else
printf("Set parameter op failed. (no memory?)\n");
```

SEE ALSO

[QGo](#) , [QGetParameterOp](#) , [QRequestPrivilegeOp](#)

1.52 qgetparameterop

amarquee.library/QGetParameterOp amarquee.library/QGetParameterOp
NAME

QGetParameterOp - Get the value of miscellaneous parameters.

SYNOPSIS

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

```
LONG QGetParameterOp(struct QSession * session, char * paramName)
```

FUNCTION

This function lets your client get the value of certain environment variables or other miscellaneous parameters on the server machine.

Note that your client will need `QPRIV_ADMIN` status to get some of these variables.

NOTE

This function requires v45 of amarquee.library, and v1.45 of AMarqueued.

INPUTS

session - The session for which you wish to get a parameter.

paramName - The keyword corresponding to the parameter you wish to get. Keywords, and who they may be retrieved by,

are described in this [table](#) .

Note that parameter names are case sensitive!

RESULTS

Returns the assigned ID number of the access operation, or 0 to indicate failure (due to memory shortage).

If you did not have permission to retrieve the parameter you indicated, a `QERROR_UNPRIVILEGED` message will be sent back from the server.

If you specified an unknown paramName, a `QERROR_MALFORMEDKEY` will be sent back from the server.

Otherwise, the parameter's value will be returned to you in a `QMessage` that has `qm_Path` set to the parameter's name, and `qm_Data` set to the parameter's value (all values will be sent as NUL-terminated ASCII strings).

EXAMPLE

LONG transID;

/ Find out who is banned from the server. */*

```
if (transID = QGetParameterOp(session, "AMARQUEED_BANNED"))
```

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

```
else
```

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

SEE ALSO

[QGo](#) , [QSetParameterOp](#) , [QRequestPrivilegeOp](#)

1.53 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);
}
```

AREXX EXAMPLE

```
message = GetNextQMessage(session, -1, 'SIGBREAKF_CTRL_C|SIGBREAKF_CTRL_F')
if (message > 0) then do
say "QMessage received-----"
say "Status: " || GetQMessageField(message, 'Status') || " (" || QErrorMessage(GetQMes
/* ... */
call FreeQMessage(session, message)
end
```

SEE ALSO

[QMessages](#)

1.54 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 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.

QGOF_NOTIFY - If set, the client TCP thread will notify your application via a QMessage as soon as it has finished sending the data in this transaction.

The returned QMessage will have a qm_Status of QERROR_SEND_DONE, and a qm_ID equal to the return value of this function. (This flag was introduced in v47 of amarquee.library, and will be ignored by earlier versions)

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(0L))
printf("Go succeeded, was given id #%li\n",goID);
else
printf("Go failed. (no memory?)\n");
```

AREXX EXAMPLE

```
LONG goID;
/* ... do various Q*Op() calls ... */
goID = QGo('QGOF_SYNC|QGOF_NOTIFY')
if (goID > 0) then
say "Go succeeded, was given id #" || goID
else
say "Go failed. (no memory?)"
SEE ALSO
QMessages
```

1.55 installdaemon

Note: These are instructions for installing a daemon program written using amarquee.library, NOT for installing the "AMarqueued" 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.

1.56 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.

AMarqueeDebug.rexx - Same thing as AMarqueeDebug, but written in ARexx.

AMarqueeDebugMultiThread - Just like AMarqueeDebug, but with an
extra thread thrown in to demonstrate
the use of [QDetachSession](#) and [QReattachSession](#) .

KillClients.rexx - A utility that allows you to kick AMarquee clients
off the AMarquee daemon running on localhost. Note
that you must set the env variable AMARQUEED_KILLCLIENTS
appropriately to give KillClients permission to do this,
or it won't work. ("/localhost/#?" is a good setting
to use)

SysMessage.rexx - A utility that lets you send system messages to
AMarquee clients logged into your machine's AMarqueed
server. (You'll need [QPRIV_SENDSYSMESSAGE](#) access to

do this)

AMarqueeHost - Like AMarqueeDebug, only instead of initiating a connection, it uses `QNewHostSession` to await an incoming connection on a specified port.

AMarqueeHost.rexx - Like AMarqueeHost, but written in ARexx.

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.

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.

StreamGen - Tests the `QStreamOp()` synchronization. Have this program logged into your AMarqueed server, and then connect one or more StreamCheck programs to the same server.

StreamCheck - Tests the `QStreamOp()` synchronization. Log these into an AMarqueed server where StreamGen is running. They should count up numbers and not report any skipped numbers.

1.57 Thanks

Thanks go to the following people:

Oliver Hotz, for hosting the first full-time AMarquee server site.

Thomas Steinbichler, for hosting the second (and current) full-time AMarquee server site (at qamitrack.tibb.at, 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.

Raj, for helping with the Inet225 versions of the code.

Petter Nielsen, Anders Jakobsen, and Dean Husby for helping with the ARexx support.

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

1.58 faq

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

A: Right now there is qamitrack.tibb.at, 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 Netris, QAmiTrack, QSendFile, 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.
- Your TCP stack is configured wrong, or you are using the wrong flavor of amarquee.library (e.g. the Inet225 version under AmiTCP, or vice versa--do a "version libs:amarquee.library file full" to find out which flavor you have installed)

Q: Why do I keep getting disconnected from the AMarquee server?

A: Again, it could be any of several reasons:

- The server shut down.
- A client with KILLCLIENT privilege has killed your connection to the server. (Such a client would usually be an admin tool being operated by the server owner)
- The TCP connection was so slow that the server thought you had crashed.
- The TCP connection was so slow that your client thought the server had crashed.
- Your TCP stack had been shut down.
- The server computer ran out of memory.
- Your computer ran out of memory.
- You and the server are running incompatible versions of AMarquee. Specifically, AMarqueued servers before v1.45 may have trouble with amarquee.library v45 and above.
- A second AMarquee client has logged into the AMarquee server, and it has the same hostname and program name as your client program. AMarquee enforces the uniqueness of the hostname/programName pair on any given server computer, so if a duplicate program logs in, the original is logged out.

1.59 The ground was littered with squashed bugs...

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

1.47 : (Public Release 2/20/97) (amarquee.library v47)

- Added a QGOF_NOTIFY flag to **QGo** ().
- Added the **QDetachSession** () and **QReattachSession** () functions to amarquee.library.
- Added AMarqueeDebugMultiThreaded.c to the testPrograms

directory, as a test/demonstration of the new functions.

- Optimized the transport code somewhat, to cut down on the number of memory allocations, and the amount of memory used.

- Added Session.h to the distribution. Session.h contains a C++ "wrapper" class named **Session**, that represents a QSession in a nice object-oriented manner.

- Added AMarqueeDebug.cpp to the **Example Clients** directory.

- * Added the link to the **QErrorMessage** documentation that was overlooked before in the **API listing**.

- * Fixed the parsing of arguments in the ARexx version of **QGo** ().

1.46 : (Public Release 12/7/97) (amarquee.library v46)

- **ARexx support** ! amarquee.library is now usable by ARexx scripts.

- Added the **QGetAndSubscribeOp** () function to amarquee.library.

- Added amarqueedebug.rexx, amarqueehost.rexx, sysmessage.rexx, and killclients.rexx to the **sample programs** directory.

- * Fixed a bug in AMarquee that prevented **QRenameOp** events from being broadcast to other clients correctly--the deletion of the old name would be sent, but not the creation of the new one.

- * **QNewHostSession** () was broken under Inet225. Fixed it. (Thanks to Raj for reporting this!)

- * Fixed a minor bug that would cause an unwarranted update message to be sent to any client that had received a **QMessageOp** message from a client whose root node it was monitoring via **QSubscribeOp**.

1.45 : (Public Release 11/2/97) (amarquee.library v45)

- Added Inet225 versions of amarquee.library and AMarquee.

Now all amarquee.library based programs can run, without modification, on both AmiTCP and Inet225 systems!

Furthermore, Inet225 systems can be AMarquee servers.

- Added Pascal headers and a link library, AMarquee.lib.

Now amarquee.library can be used by PCQ programs!

- Extended the Installer script to install to Inet225, and to (optionally) install the PCQ specific files.

It also helps you set the new env variables (described below)

- Added the **QRequestPrivilegesOp**, **QReleasePrivilegesOp**, and **QKillClientsOp** functions to amarquee.library.

With these functions, favored clients can request special abilities from the server.

- Added the **AMARQUEED_KILLCLIENTS**, **AMARQUEED_ADMIN**, **AMARQUEED_SENDSYSMESSAGES** and **AMARQUEED_ALLPRIVILEGES** env variables to

AMarqueued, to allow the site administrator to specify who can get special abilities.

- Added the `AMARQUEED_MAXQUEUEDMESSAGES` env variable to AMarqueued. This setting allows you to have AMarqueued automatically disconnect clients who are getting too far behind in their duties (due to e.g. a bad TCP connection to their client), so that they don't eat up all your memory.

- Added the `QGetParameterOp` and `QSetParameterOp` functions to `amarquee.library`, so that client programs may get or set certain `parameters` on the server.

- Added client-to-server ping logic to `amarquee.library`.

Now your client should know within 10 minutes that the server has crashed/gone offline, even if it didn't go down "gracefully", and your connection was idle the whole time.

* Fixed a typo in the `.fd` file that was causing `QErrorMessage` to return incorrect results.

* Fixed a bug in the server-to-client ping logic. The server should detect "dead" client connections somewhat more quickly now.

* Fixed a bug in the access-control logic. Before, `QSetAccessOp` kept other clients from "seeing" even the root node of its client, rather than just the contained data--making the client effectively invisible to other clients.

* Server now sends a `QERROR_UNIMPLEMENTED` message when it gets a packet it doesn't know the type of, rather than disconnecting the client. This should ease compatibility problems in the future.

1.44 : (Public Release 6/3/97) (`amarquee.library` v44)

- Added the `QNumQueuedPackets` and `QNumQueuedBytes` functions to `amarquee.library`.

- Added the `QErrorMessage` function to `amarquee.library`

- Added some additional `QERROR_*` codes, and added a return value to `QFreeSession` so that it can return them.

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 residenting will still work even if AMarquee's PURE bit isn't set.

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

- Both AMarquee 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.

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

- Changed AMarquee'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 AMarquee, 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.

1.60 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!

- Server linking?

- Write some more cool AMarquee client programs.

- UDP support

- An AMarqueued admin tool, based on the new admin functions put into amarquee.library v45.

1.61 unnamed.1

1.62 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: " "
 - AMarquee's "program name" based access control is insecure, since any program can log in with any login name. (And, if the evil hackers are smart enough to be able to do packet hostname spoofing, than the "host name" based access control becomes insecure too) Since AMarqueed never reads from or writes to disk, and does sanity checks for on the input from TCP clients, this isn't quite as bad as it sounds...
 - **QNewServerSession** () won't work with the inet225 version of amarquee.library.
-

1.63 otherprogs

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

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/QAmiTrack1.92.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/ARemote1.01B.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)

Netris - A four-player Internet Tetris game. Play Tetris against three of your friends! Features customizable, shareable sound effects, and in-game chat lines for taunting. :)

Uses the AMarquee system for TCP communication.

(comm/net/Netris1.15.lha)

QSendFile - An AMarquee based program to let you send files to another Amiga. I wrote this so that I could transfer files to my Mom's Amiga without her having to know how do anything other than click "Okay" on her end.

(comm/net/QSendFile1.0B.lha)

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)