# QCOM Reference Guide

Lars Wirfelt 2002-06-10

# Table of Contents

Overview of Qcom ....5
- Message Bus....5
- Qcom ....6
  - The Bus ....6
  - The Queue ....7
  - The message ....7
- Summary of calls ....7

Queues ....9
- Private Queue....9
  - Creating a private queue ....10
  - Attaching a private queue ....11
- Forwarding Queue ....11
  - Creating a forwarding queue ....12
  - Binding to a forwarding queue ....12
  - Unbinding from a forwarding queue ....13
  - Deleting a forwarding queue ....13
  - Exiting an application ....13
- Broadcast queue ....13
  - Creating a broadcast queue ....13
- Event queue ....14
  - Creating an event queue ....14
  - Signalling an event queue ....14
  - Waiting on an event queue ....15
  - Binding to an event queue ....15
  - Query an event queue ....15
- Special queues ....16
  - qcom_cNQid ....16
  - qcom_cQnetEvent ....16
  - qcom_cQapplEvent ....16

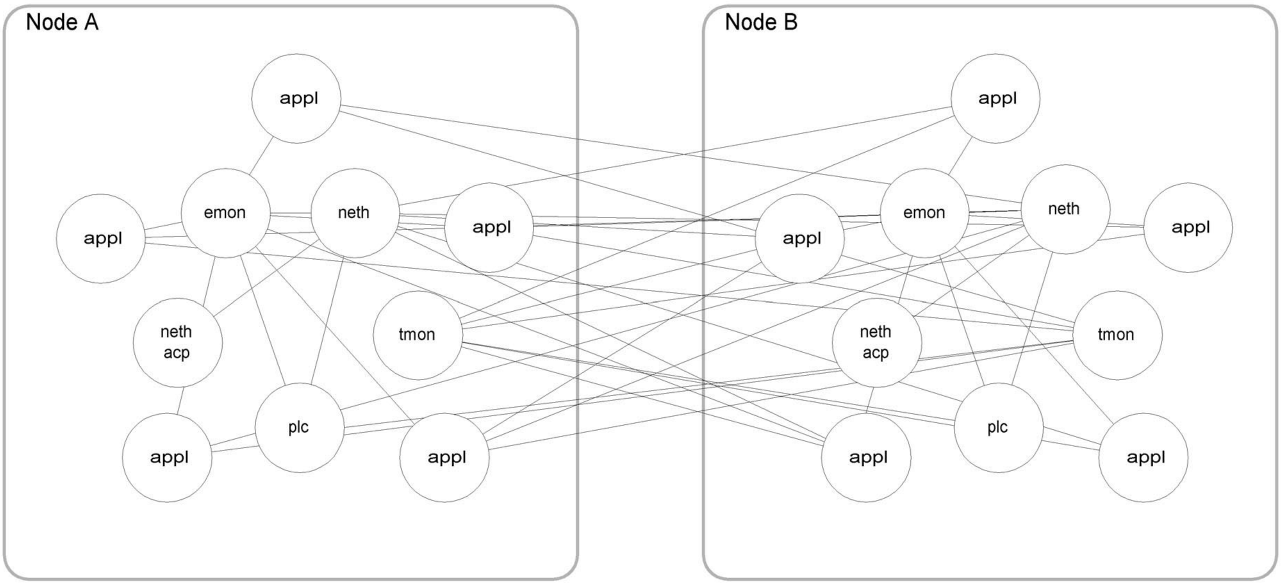
Using the Qcom API ....17
- Types ....17
  - qcom_sQid ....17
  - pwr_tNodeId....18
  - qcom_sAid ....18
  - qcom_sAppl ....18
  - qcom_sEvent ....18
  - qcom_sQattr ....19
  - qcom_sType ....19
  - qcom_sPut ....19
  - qcom_sGet ....20
  - qcom_sNode ....20
- Connection calls ....21
  - Connecting to Qcom ....21
  - Exiting from Qcom ....21
  - Creating a queue ....21
  - Deleting a queue ....22
- Sending and receiving ....22

Using qcom_Put and qcom_Get ....................................................................................22
Using qcom_Request and qcom_Respond ...................................................................23
Buffer allocation ........................................................................................................24
Qmon ..............................................................................................................................26
Network Status ..............................................................................................................26
Configuration.................................................................................................................27
The Bus Identity ...........................................................................................................27
The Node File ...............................................................................................................27

# Overview of Qcom

A Proview system consists of a number of applications distributed on a number of nodes in a network. Each application has to communicate with other applications on the same node as well as with applications on other nodes. A common way to do this is using point to point communication, using for example TCP/IP socket communication. On a typical Proview system this would result in a great number of socket pairs.

**Figure 1-1 Point to point communication**




Each application would have to care about things like connections and disconnections, handling of nodes disappearing and reappearing, segmentation of large messages and more.

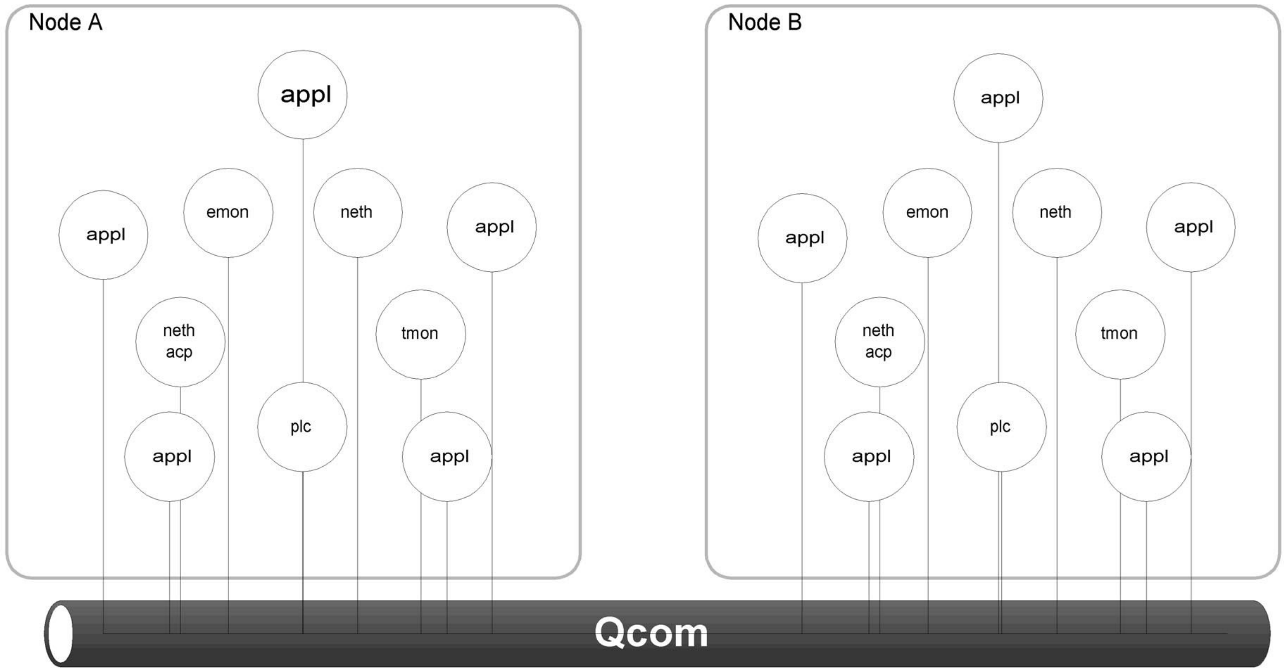
## *Message Bus*

Another way to solve this is using a message bus, a software component where

- all network events and work units (data) are packaged into messages,
- messages can be of variable size and can be categorized by user definable message types,
- messages preserve the "write"ӿ(i.e. record) boundaries of the sending application,
- applications have a single attachment point to the bus where all communication (i.e. messages) to other processes are funneled,
- an application communicate with another application, either local or remote, using the same API (although the implementation could be quite different),
- the implementation is host and network backbone independent, and
- applications connected to the message bus can communicate with any other connected

application, without formal connection sequence routines required for each partner.

**Figure 1-2 Communication with a message bus**




A bus topology is inherently simpler to attach and control. This makes peer to peer communication simple and efficient. To summarize the message bus, provides flexible services and methods for distributed applications to communicate with one another and share data.

# Qcom

Qcom, Queue Communication, is an implementation of the Message Bus architecture. Qcom is a combination of

- an interprocess message routing mechanism
- process wait and wakeup mechanism
- a monitor (daemon) to distribute messages between nodes

Qcom isolates the application programmer from having to concern themselves about details of an interprocess communication implementation. Traditional communication implementations are machine architecture and operation system specific, and can require considerable system expertise. By isolating application code from the actual communications mechanisms, the system can be easily upgraded to use more efficient techniques as the hardware and operating system software evolves. These techniques can be incorporated into Qcom routines without effecting the user code.

Initialization of Qcom is done as part of Proview startup procedures.

## The Bus

Several buses can coexist on the same node, but it is not possible to communicate between buses. This can be used to test a system at the same time as the production system is running. Start a new bus and run tests using this bus.

### The Queue

A central concept of Qcom is the queue. An application owns one or many queues.

Applications can write to a queue either on the same node or to a queue on a remote node. Each queue has a globally unique identity and other applications can, knowing the identity (it does not have to know the location of the queue), send messages to any queue.

A queue can hold a number of unread messages and the application owning the queue can read the messages in its own pace.

There are different kinds of queues in Qcom:

- private queue - messages written to the queue is read by the application owning the queue,
- forwarding queue - a number of queues can be bound to a forwarding queue, and messages written are forwarded to all bound queues, a convenient way to send a message to a group of applications,
- broadcast queue - like a forwarding queue but messages are also sent to all other nodes on the bus,
- event queue - used to synchronize applications.

### The message

Applications communicate by sending messages. Each message can be assigned a type and a sub type. The message type is a way of grouping categories of messages while the message sub type is used to identify messages within the category.

# *Summary of calls*

The application interface Qcom consists of

`qcom_Init(), qcom_Exit()`
to connect and disconnect to Qcom,

`qcom_CreateQ(), qcom_AttachQ(), qcom_DeleteQ()`
to handle queues,

`qcom_Put(), qcom_Get(), qcom_Request(), qcom_Reply()`
to send and receive messages,

`qcom_Alloc(), qcom_Free()`
to allocate and free message buffers,

`qcom_Bind(), qcom_Unbind()`
to control binding to forwarding and broadcast queues,

`qcom_SignalAnd(), qcom_SignalOr(), qcom_WaitAnd(), qcom_WaitOr(),`
`qcom_EventMask()`
to handle events,

`qcom_AidCompare(), qcom_AidIsEqual(), qcom_AidIsNotEqual(), qcom_AidIsNotNull(),`

`qcom_AidIsNull()`
to compare application identities,

`qcom_MyBus(), qcom_MyNode(), qcom_NextNode()`
to get information about the bus and nodes,

`qcom_QidCompare(), qcom_QidIsEqual(), qcom_QidIsNotEqual(), qcom_QidIsNull(), qcom_QidIsNotNull()`
to compare queue identities, and

`qcom_QidToString()`
to convert a queue identity to string.

# Queues

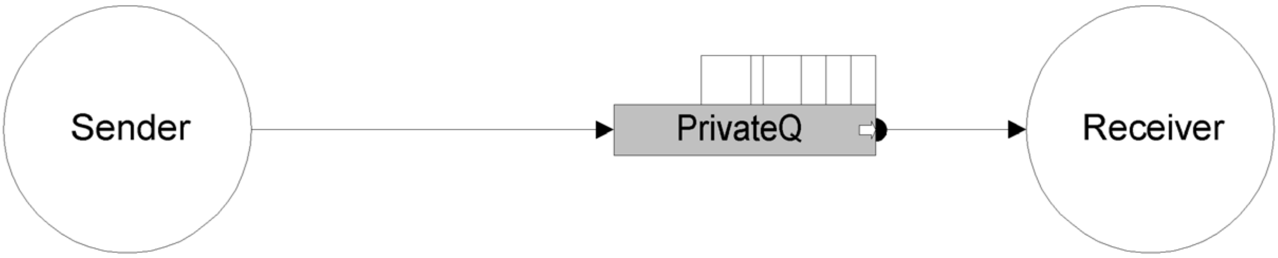
This chapter describes the different kinds of Qcom queues and how to use them.

## *Private Queue*

A private queue is created and owned by one application (process). Only this application can read from the queue. Any application can write to the queue, either directly or via a forwarding queues. The application can be threaded, Qcom is thread safe.

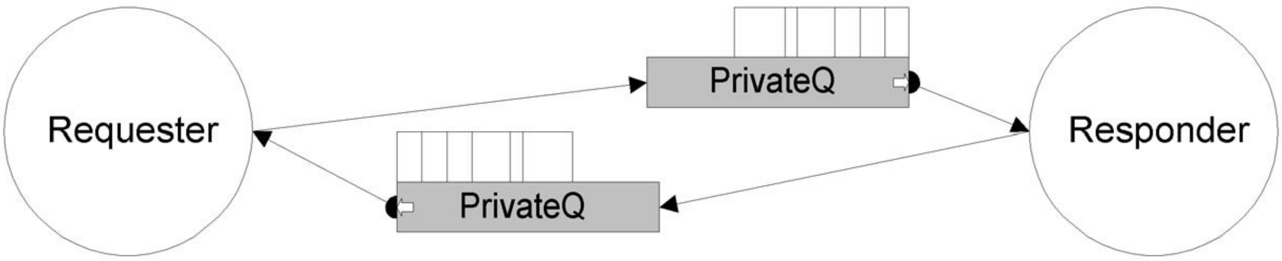
A private queue can also be created without ownership. An application can later on attach to the queue and in that way take ownership of the queue. Only Proview internal applications can create such non-owned queues.

**Figure 2-1 A private queue**



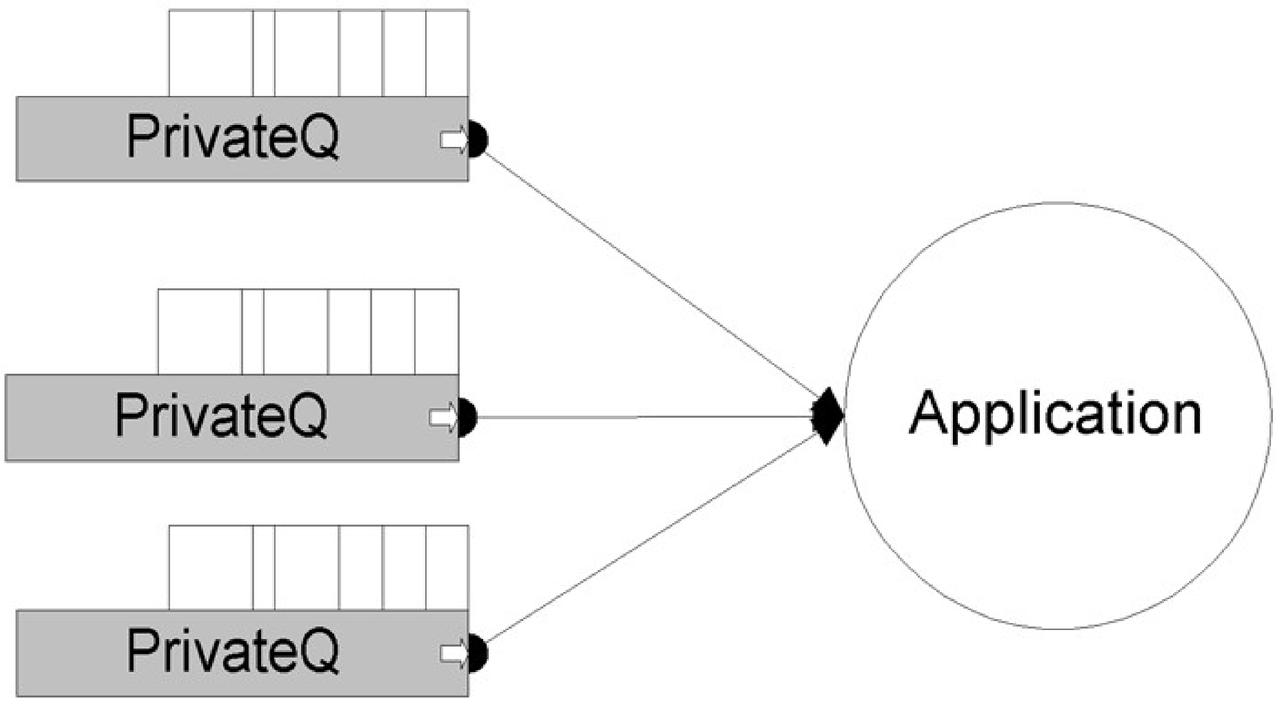

The receiver application owns and reads from a private queue. The sender application can write to this queue. Note that this is one way communication, for duplex communication you need two queues, one for each application.

**Figure 2-2 Duplex communication**




An application can have many private queues.

**Figure 2-3 Many private queues**




An application using GDH and MH_OUTUNIT will have two private queues, implicitly created at initialization of the respective interface, and then any number of explicitly created queues.

## Creating a private queue

A private queue is created using the qcom_CreateQ call.

```
pwr_tStatus sts;
qcom_sQid myQ = qcom_cNQid;
qcom_sQattr attr;
char *name = “myQ”;

attr.type = qcom_eQtype_private;

if (!qcom_CreateQ(&sts, &myQ, &attr, name)) {
  // report error
}
```

In this case the queue identity “myQ” is initialized to the null queue identity, and Qcom will assign a random, unique, queue identity. To create a queue with a predefined known identity, “myQ” must be initialized to the wanted identity before calling qcom_CreateQ.

```
pwr_tStatus sts;
qcom_sQid myQ = {0, aPredefinedKnownQid};
qcom_sQattr attr;
char *name = “myQ”;

attr.type = qcom_eQtype_private;

if (!qcom_CreateQ(&sts, &myQ, &attr, name)) {
  // report error
}
```

If “name”□ is a null pointer the queue will get the name “unknown name”.

If “attr” is a null pointer the queue type will default to private.

### Attaching a private queue

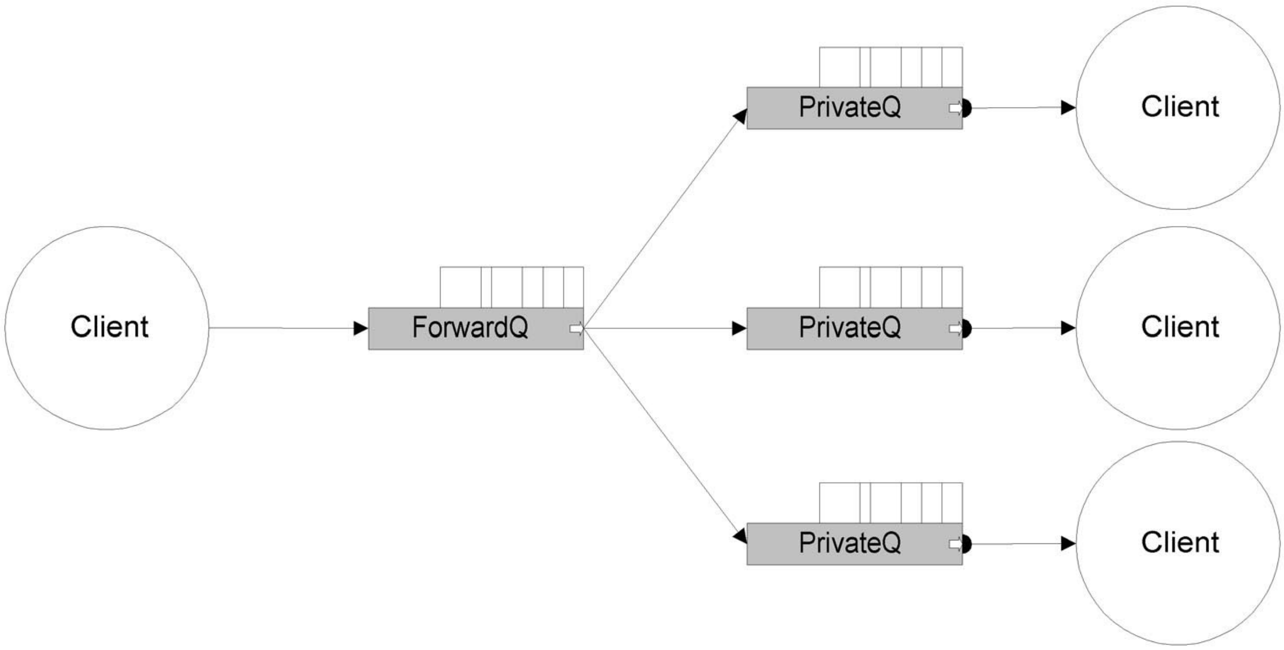
A private queue is attached using the qcom_AttachQ call.

# *Forwarding Queue*

A forwarding queue is a convenient way to send one message to a group of applications, a kind of selective broadcast. Applications that wants to receive messages sent to a forwarding queue do so by binding one or more of its private queues to the forwarding queue.

Every message written to a forwarding queue is forwarded to all queues bound to the forwarding queue at that specific moment. Messages are not saved in the forwarding queue, so an applications binding to a forwarding queue will only receive messages written to the forwarding queue after the bind call.

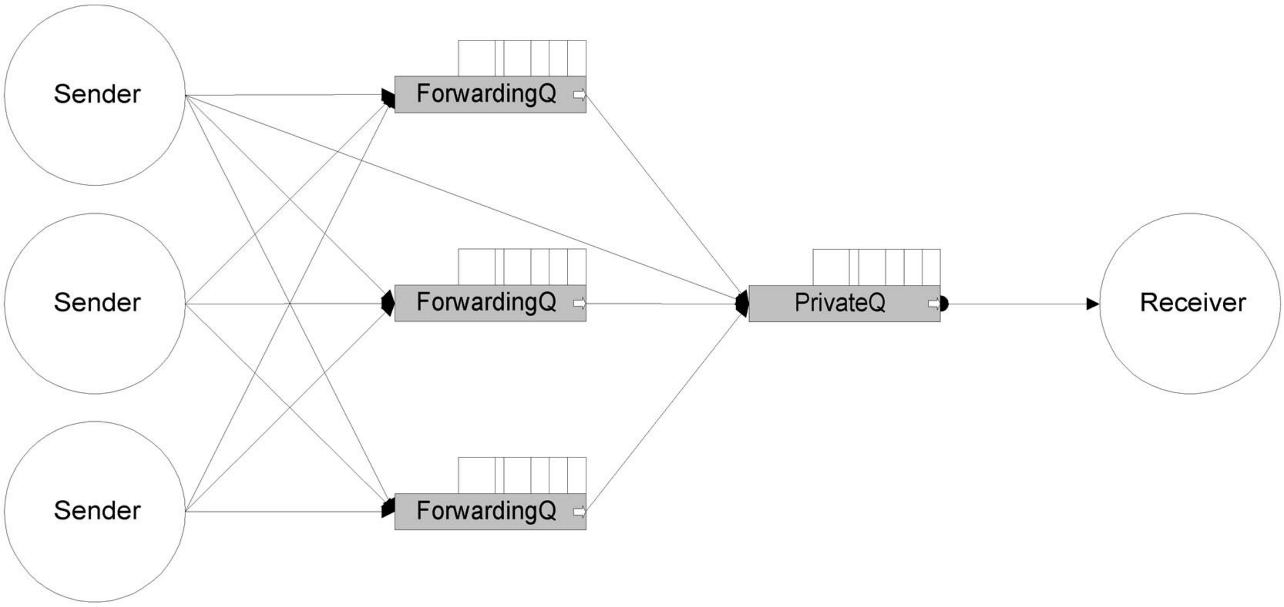
**Figure 2-4 Forwarding queue**




An application cannot read from a forwarding queue directly. The only way is to bind to the forwarding queue.

A private queue can be bound to many forwarding queues.

**Figure 2-5 Forwarding queues**




## Creating a forwarding queue

```
pwr_tStatus sts;

qcom_sQid forwardQ = qcom_cNQid;
qcom_sQattr attr;
char *name = "aForwardingQ";

attr.type = qcom_eQtype_forward;

if (!qcom_CreateQ(&sts, &forwardQ, &attr, name)) {
  // report error
}
```

If "name" is a null pointer the queue will get the name "unknown name".

A forwarding queue is owned by the application that created it. If this application exits, the forwarding queue will disappear and other queues bound to this queue will be unbound.

## Binding to a forwarding queue

Only a private queue can bind to a forwarding queue, and only a forwarding queue can be bound to a private queue. A queue is bound to a forwarding queue using the qcom_Bind call.

```
pwr_tStatus sts;

qcom_sQid myQ;
qcom_sQid forwardQ;

if (!qcom_Bind(&sts, &myQ, &forwardQ)) {
  // handle error
}
```

After this all messages sent to "forwardQ" is forwarded to "myQ".

## Unbinding from a forwarding queue

To unbind from a forwarding queue use the qcom_Unbind call.

```
pwr_tStatus sts:
qcom_sQid myQ;
qcom_sQid forwardQ;

if (!qcom_Unbind(&sts, &myQ, &forwardQ)) {
  // handle error
}
```

Messages, originally sent to the forwarding queue, pending on the private queue, will still be left pending, but no new messages will be forwarded.

## Deleting a forwarding queue

If a forwarding queue i deleted, all queues bound to it will first be unbound. Pending messages will not be deleted.

## Exiting an application

If an application with a queue bound to forwarding queues exits, the queue will be unbound during exit clean up.

If the application owns forwarding queues, all queues bound to the forwarding queue will be unbound and then the forwarding queue will be deleted.

# *Broadcast queue*

A broadcast queue is like a forwarding queue with the addition that messages except from being forwarded on all bound queues also are forwarded to all other known nodes. When a broadcast message arrives at a remote node, Qcom looks for a broadcast queue with the same queue index. If such a queue exists the message will be written to all queues bound to the remote broadcast queue. Binding and unbinding to a broadcast queue is done in the same way as with forwarding queues.

## Creating a broadcast queue

```
pwr_tStatus sts;
qcom_sQid broadcastQ = {0, cQindex};
qcom_sQattr attr;
char *name = "aBroadcastQ";

attr.type = qcom_eQtype_broadcast;

if (!qcom_CreateQ(&sts, &broadcastQ, &attr, name)) {
  // handle error
}
```

If “name” is NULL the queue will get the name “unknown name”.

Notice that the queue identity is initialized with a predefined known value. The whole idea with a broadcast queue is that other applications know about its existence.

# *Event queue*

An event queue is used for applications to synchronize on different events. It has the forwarding queue capabilities, but also some extra characteristics.

An event queue has a 32-bit bitmask and there are a number of Qcom calls to query and manipulate the bitmask.

An application can signal an event on the event queue, it can bind to an event queue, and it can wait on an event queue.

Typically an event queue is used by a group of applications, each of which has to agree on the meaning of each single bit in the bitmask.

## Creating an event queue

```
pwr_tStatus sts;
qcom_sQid eventQ = {0, cEventQ};
qcom_sQattr attr;
char *name = "anEventQ";

attr.type = qcom_eQtype_event;

if (!qcom_CreateQ(&sts, &eventQ, &attr, name)) {
  // handle error
}
```

Notice that the queue identity is initialized with a predefined known value. The whole idea with an event queue is that other applications know about its existence.

## Signalling an event queue

An application can signal an event queue using qcom_SignalOr() or qcom_SignalAnd() calls.

```
pwr_tStatus sts;
qcom_sQid eventQ = {0, cEventQ};

int mask = 1 << 4;

if (!qcom_SignalOr(&sts, &eventQ, mask)) {
  // handle error
}
```

With qcom_SignalOr the bit mask associated with the event queue, is bitwise ored with the value of "mask", and with qcom_SignalAnd the associated mask is anded with the value of "mask".

Applications waiting on the event queue will be woken if the new event mask matches their wait condition.

## Waiting on an event queue

An application can wait on an event queue using qcom_WaitOr() or qcom_WaitAnd() calls.

```
pwr_tStatus sts;
qcom_sQid myQ;
qcom_sQid eventQ = {0, cEventQ};
int mask = myEvent;

if (!qcom_WaitOr(&sts, &myQ, &eventQ, mask, qcom_cTmoEternal)) {
  // handle error
}
```

In this case the application will sleep until either an event causing the bit mask, associated with the queue “eventQ”, to match the mask in the wait call, or, a message is written to “myQ” or any queues bound to “myQ”. In this way an application can wait both on messages and an event. To be awaken only on events the application can create a new queue to be used only for this purpose.

## Binding to an event queue

Another way to be notified of events is to bind a queue to an event queue.

When an application signals the event queue, Qcom will generate a message and write it on all bound queues. The message will have message base type qcom_eBtype_event and sub type equal to the queue index of the event queue.

See “qcom_sEvent” for more information.

```
pwr_tStatus sts;
qcom_sQid myQ;
qcom_sQid eventQ = {0, cEventQ};
int mask = myEvent;
qcom_sGet get;

if (!qcom_Bind(&sts, &myQ, &eventQ)) {
  // handle error
}

for (::) {
  get.data = NULL;
  if (!qcom_Get(&sts, &myQ, &get, qcom_cTmoEternal)) {
    //handle error
  }
  switch (get.type.b) {
  case wantedEventType:
    qcom_sEvent *ep = (qcom_sEvent *)&get.data;
    if (ep->mask & wantedMask) {
      // do something appropriate
    }
    break;
  case ...
  }
  qcom_Free(&sts, &get.data);
}
```

## Query an event queue

An application can query the current mask of an event queue without synchronizing on it.

```
pwr_tStatus sts;
qcom_sQid eventQ = {0, cEventQ};
```

```
if (qcom_EventMask(&sts, &eventQ) & wantedEvent) {
  // do something appropriate
}
```

# *Special queues*

## qcom_cNQid

The null queue, i.e. no queue at all.

## qcom_cQnetEvent

A queue bound to this forwarding queue will receive network status events.

See “Network Status” on page 4-1 for more information.

## qcom_cQapplEvent

A queue bound to this forwarding queue will receive messages with application connect and disconnect events.

See “qcom_sAppl” on page 3-2 for more information.

# Using the Qcom API

To use the Qcom Application Programmerâ€™s Interface include the rt_qcom.h in files calling Qcom.

```
#include "rt_qcom.h"
#include "rt_qcom_msg.h"
```

Linking is done using the ordinary libpwr_* libraries.

## *Types*

### qcom_sQid

```
typedef struct {
  qcom_tQix qix;
  pwr_tNodeId nid;
} qcom_sQid;
```

Every queue within a Qcom bus is uniquely identified by a queue identity, used for identifying the target for sending a message.

- **qix** intra-node queue index.
- **nid** node identity, if set to zero, delivery will default to the local node, if non-zero Qcom will pass the message to the remote Qcom node for delivery.

Queue identities are assigned in two ways, permanent and temporary identities. Queues that needs a predefined known addresses uses a qix where the most significant bit (the sign bit) is set, giving the range 0x80000000 - 0xffffffff. Of these the first 1000 are reserved by the system, 0x800003e8, and the rest are open for applications to use. Note however that there is no reservation system in Qcom for these addresses.

Queue identities may also be allocated as temporary queue identities. This does not imply that the application is temporary, but that the assignment of the identity is done dynamically at run-time. Any application that requires multiple copies of a program to run will usually be declared as a temporary process to allow a queue id to be assigned dynamically. Qcom uses qix in the range 0x00000001 - 0x7fffffff for temporary queue identities.

The following Qcom routines are used for comparing queue identities.

```
int          qcom_QidCompare(const qcom_sQid*, const qcom_sQid*);
pwr_tBoolean qcom_QidIsEqual(const qcom_sQid*, const qcom_sQid*);
pwr_tBoolean qcom_QidIsNotEqual(const qcom_sQid*, const qcom_sQid*);
pwr_tBoolean qcom_QidIsNull(const qcom_sQid*);
pwr_tBoolean qcom_QidIsNotNull(const qcom_sQid*);
```

To convert a queue identity to string format.

```
char * qcom_QidToString(char*, qcom_sQid*, int);
```

## pwr_tNodeId

Every node within one Qcom bus i uniquely identified by a node identity. This identity is also used by other parts of Proview.

## qcom_sAid

```
typedef struct {
  qcom_tAix aix;
  pwr_tNodeId nid;
} qcom_sAid;
```

- **qix** intra-node application index,
- **nid** node identity

```
static const qcom_sAid qcom_cNAid = {0, 0};
```

Every application connecting to the Qcom bus will get a unique application identity. This identity is used to identify the source which generated a message. The application identity is also shown in log messages in the error log.

The following Qcom routines are used for comparing application identities.

```
int          qcom_AidCompare(const qcom_sAid*, const qcom_sAid*);
pwr_tBoolean qcom_AidIsEqual(const qcom_sAid*, const qcom_sAid*);
pwr_tBoolean qcom_AidIsNotEqual(const qcom_sAid*, const qcom_sAid*);
pwr_tBoolean qcom_AidIsNotNull(const qcom_sAid*);
pwr_tBoolean qcom_AidIsNull(const qcom_sAid*);
```

## qcom_sAppl

```
typedef struct {
  qcom_sAid aid;
  pid_t pid;
} qcom_sAppl;
```

An application can receive notification about other applications connecting or disconnecting from Qcom. To receive application events at least one queue has to be bound to the forwarding queue qcom_cQapplEvent. Application event are received as messages with basic type qcom_eBtype_qcom and subtypes qcom_eStype_applConnect and qcom_eStype_applDisconnect. The data part of the message contains a qcom_sAppl.

- **aid** is the identity of the application that signaled the event queue
- **pid** is the process identity of the application

## qcom_sEvent

```
typedef struct {
  qcom_sAid aid;
  pid_t pid;
  int mask;
} qcom_sEvent;
```

If an event queue is bound to other queues, a message will be generated each time the queue is signalled. The data part of such a message is of type qcom_sEvent.

- **aid** is the identity of the application that signaled the event queue
- **pid** is the process identity of the application
- **mask** is the content of the associated event mask after the signal

## qcom_sQattr

```
typedef struct {
  qcom_eQtype type;
  unsigned int quota;
} qcom_sQattr;
```

A queue has some attributes that can be set by an application at queue creation time.

- type — to specify what kind of queue is to be created<br>qcom_eQtype_private<br>qcom_eQtype_forward<br>qcom_eQtype_broadcast<br>qcom_eQtype_event
- quota — to specify the maximum number of pending messages on a queue

## qcom_sType

```
typedef struct {
  qcom_eBtype b;
  qcom_eStype s;
} qcom_sType;
```

Messages can be categorized in base type and sub type. Basic types in the range 0-1000 are reserved by the system and the rest are free for application us.

## qcom_sPut

```
typedef struct {
  qcom_sQid reply;
  qcom_sType type;
  unsigned int size;
  void *data;
} qcom_sPut;
```

Used to describe a message to be sent.

- **reply** identity of queue to receive a reply (An application wanting an answer on a message uses this filed to indicate on what queue it will read the answer.),
- **type** type of message
- **size** size of the “data” part of the message
- **data** pointer to data buffer to be sent

## qcom_sGet

```
typedef struct {
  qcom_sAid sender;
  pid_t pid;
  qcom_sQid receiver;
  qcom_sQid reply;
  qcom_sType type;
  qcom_tRid rid;
  unsigned int maxSize;
  unsigned int size;
  void *data;
} qcom_sGet;
```

Gives information on the message just received.

- sender application identity of sender
- pid process identity of process running the application
- receiver identity of queue that received the message
- reply identity of queue to receive a reply
- type type of message
- rid request identity, used to match a request - reply pair
- maxSize used when using private buffers, to indicate the size of the receive buffer
- size size of the “data” part of the actually received message
- data pointer to data buffer received

## qcom_sNode

```
typedef struct {
  pwr_tNodeId nid;
  qcom_mNode flags;
  char name[80];
  qcom_eOS os;
  qcom_eHW hw;
  qcom_eBO bo;
  qcom_eFT ft;
} qcom_sNode;
```

An application can receive notification of network status changes. To receive network events at least one queue has to be bound to the forwarding queue qcom_cQnetEvent. Network event are received as messages with basic type qcom_eBtype_qcom and subtypes:

- **nid** node identity
- **flags** the status of the connection to node
  qcom_mNode_initiated
  qcom_mNode_connected

qcom_mNode_active

- **name** name of node
- **os** the operating system run on the node
- **hw** the hardware platform of the node
- **bo** byte order
- **ft** floating point format

- qcom_eStype_linkConnect,
  a node has established connection
- qcom_eStype_linkDisconnect,
  a node has disappeared, normally happens only when a node is restarted
- qcom_eStype_linkActive,
  communication with the node is working smoothly
- qcom_eStype_linkStalled,
  requests to the node has not been answered within the stipulated time

The data part of the message contains a qcom_sAppl.

# Connection calls

## Connecting to Qcom

Before using Qcom an application must connect to Qcom.

```
pwr_tBoolean qcom_Init(pwr_tStatus *sts, qcom_sAid *aid, char *name);
```

The application has an identity and name. The identity is generated by Qcom and is returned in “aid”. If “name” is a null pointer the application will be given the name “unknown name”. Every message sent from an application contains the application identity and the identity can be read by the receiving application.

Applications using GDH, MH_APPL or MH_OUTUNIT do not have to call qcom_Init(), it is done inside the gdh_Init() and mh_OutunitConnect() calls.

## Exiting from Qcom

```
pwr_tBoolean qcom_Exit(pwr_tStatus *sts);
```

Disconnects an application from the Qcom message bus, all resources such as, queue, messages and bindings, held by the application will be released.

### Creating a queue

```
pwr_tBoolean qcom_CreateQ(pwr_tStatus *sts, qcom_sQid *myQ, qcom_sQattr *attr,
                          char *qname);
```

Create a queue. Chapter “Queues” on page 2-1 discusses different queue types and how to create them.

### Deleting a queue

```
pwr_tBoolean qcom_DeleteQ(pwr_tStatus *sts, const qcom_sQid *myQ);
```

Delete a queue and release all resources held by the queue.

# *Sending and receiving*

Sending messages is normally done with qcom_Put() and receiving with qcom_Get(). The qcom_Request() and qcom_Respond() can be used when dealing with transactions where it is essential to match a request with the right answer.

### Using qcom_Put and qcom_Get

```
void* qcom_Get(pwr_tStatus *sts, const qcom_sQid *myQ, qcom_sGet *get, int
tmo_ms);
pwr_tBoolean qcom_Put(pwr_tStatus *sts, const qcom_sQid *receiver, qcom_sPut
*put);

--- appl_a ----

qcom_sPut put;
qcom_sGet get;
char data[] = “A small question”;

...

put.reply = q_a;
put.type.b = 2001;
put.type.s = 1;
put.size = strlen(data) + 1;
put.data = data;

get.data = 0;

qcom_Put(&sts, &q_b, &put);

qcom_Get(&sts, &q_a, &get, qcom_cTmoEternal);
// use result
// Note! Do not forget to free data!
qcom_Free(&sts, get.data);

--- appl_b ----

qcom_sPut put;
qcom_sGet get;
char data[] = “A small answer”;

...
```

```
put.reply = q_b;
put.type.b = 2001;
put.type.s = 2;
put.size = strlen(data) + 1;
put.data = data;

get.data = malloc(100);
get.maxSize = 100;

qcom_Get(&sts, &q_b, &get, qcom_cTmoEternal);
// Note, do not call qcom_Free here, as the buffer was private to the
// application
qcom_Put(&sts, &get.reply, &put);
....
```

## Using qcom_Request and qcom_Respond

```
pwr_tBoolean qcom_Reply(pwr_tStatus *sts, qcom_sGet *get, qcom_sPut *put);

void* qcom_Request(pwr_tStatus *sts, const qcom_sQid *receiver, qcom_sPut*,
                   const qcom_sQid*myQ, qcom_sGet *get, int tmo_ms);
```

Imagine a situation where an application sends a request to another application.

```
qcom_Put(&sts, &q_b, &put);

qcom_Get(&sts, &q_a, &get, qcom_cTmoEternal);
```

The message is received at the target and an answer is sent, but by some reason the answer is delayed beyond the time-out in the qcom_Get call of the requester. Later on the answer arrives on the requesters queue. Then the requester does a new request.

```
qcom_Put(&sts, &q_b, &put);

qcom_Get(&sts, &q_a, &get, qcom_cTmoEternal);
```

Now qcom_Get() will return directly, but with the old answer. This could be a formally correct answer, but still an answer to another request.We have an error that could be very hard to find. To avoid this situation the applications can use qcom_Request()/qcom_Reply() instead.

```
--- appl_a ----

qcom_sPut put;
qcom_sGet get;
char data[] = "A small question";

...

put.reply = q_a;
put.type.b = 2001;
put.type.s = 1;
put.size = strlen(data) + 1;
put.data = data;
```

```
get.data = 0;

qcom_Request(&sts, &q_b, &put, &q_a, &get, qcom_cTmoEternal);
// use result
// Note! Do not forget to free data!
qcom_Free(&sts, get.data);

--- appl_b ----

qcom_sPut put;
qcom_sGet get;
char data[] = "A small answer";

...

put.reply = q_b;
put.type.b = 2001;
put.type.s = 2;
put.size = strlen(data) + 1;
put.data = data;

get.data = malloc(100);
get.maxSize = 100;

qcom_Get(&sts, &q_b, &get, qcom_cTmoEternal);
// Note, do not call qcom_Free here.

qcom_Reply(&sts, &get, &put);

....
```

The qcom_Request() call combines qcom_Put() and qcom_Get() in one call, and the application is guaranteed that at the return from qcom_Request() it either has the correct reply on the request or a time out. Internal to the qcom_Request() call, Qcom filters away any stray responses.

The qcom_Reply() call looks almost like a qcom_Put(), but the queue id is replaced with a qcom_sGet.

Applications must agree on using qcom_Request/qcom_Reply, using a qcom_Put to reply on a qcom_Request will not work.

## Buffer allocation

Internally Qcom uses a memory pool for data structures such as applications, queues, and messages. When sending a message an application can use private data, allocated on the stack, head, or static memory, or allocate data from the Qcom pool.

```
char data[100];
qcom_sPut put;

// prepare data
put.data = data;
qcom_Put(&sts, &q, &put);
```

Internally Qcom will allocate a buffer from the pool and copy user data to that buffer.

Another way is to use a buffer allocated from the pool.

```
put.data = qcom_Alloc(&sts, sizeof(data));
// prepare data
```

```
qcom_Put(&sts, &q, &put);
```

Qcom checks if the buffer is allocated in the pool or not.

The same applies when receiving a message.

```
char data[100];
qcom_sGet get;
get.data = data;
get.maxSize = sizeof(data);
qcom_Get(&sts, &q, &gut, tmo);
// use buffer data
```

The maxSize field is used to tell Qcomthemaximumsize of data to be copied to the data buffer. If the buffer is to small to hold the buffer it will be truncated and“sts” will be set to QCOM__BUFOVRUN.

To avoid copying set the data field in qcom_sGet to zero.

```
qcom_sGet get;

get.data = 0;
qcom_Get(&sts, &q, &gut, tmo);
// use buffer data
qcom_Free(&sts, get.data);
```

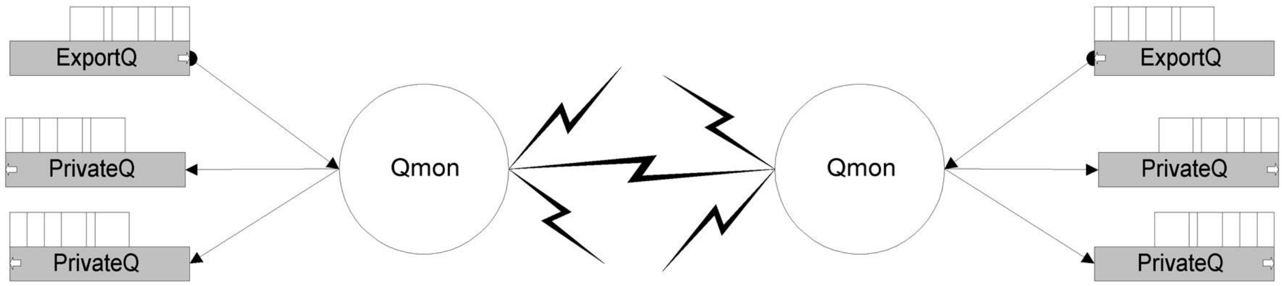
In this case the application can directly access message data in the Qcom pool. The message buffer must be freed after use.

# Qmon

Qmon, the Qcom Monitor, is responsible for communication with other Qcom nodes within a Qcom bus. Messages sent to queues on other nodes will be written to the Export queue. Qcom reads the Export queue and sends the message to the node indicated in the queue identity.

Messages received from other nodes will be written to the queue identified by the queue identity in the message. Messages to non-existing queues will be dropped.

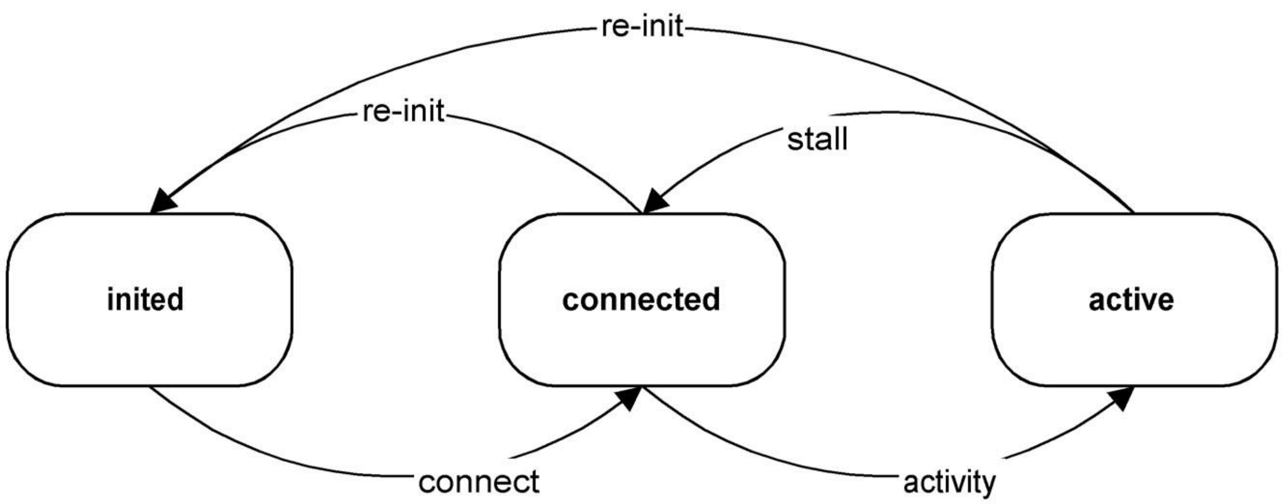
**Figure 4-1 Qmon**




## *Network Status*

While communicating with other nodes, Qmon also maintains information about each node

**Figure 4-2 States of a node**



- **inited**, the node is known but Qmon has not established communication with it.
- **connected**, communication is established but Qmon has outstanding, not answered, requests to the node.
- **active**, communication is established and flows smoothly.

For each change of status Qmon will generate a message and write it on the qcom_cQnetEvent forwarding queue.

# *Configuration*

Not much is needed to configure Qcom. Qcom is initialized and started as part of the Proview startup procedures.

## The Bus Identity

The environment variable PWR_BUS_ID must be defined and set to the bus identity.

```
--- a UNIX shell script ---
export PWR_BUS_ID=”154”

--- a VMS COM file ---
PWR_BUS_ID := 154
```

## The Node File

At startup the monitor needs to know what nodes to contact. The file $pwrp_load/ld_node_busid.dat is generated by the development environment and is read at Proview startup.

Rows beginning with # in the file are skipped.

Each row contain:

- node name
  the network name of the node
- root volume identity
- TCP/IP address
- and optionally the wanted Qmon UDP port number. If not given it will default to 55000 + <bus identity>

**Example**

```
#
#<node name> <root volume identity> <TCP/IP address> [<Qmon port number>]
#
fermat 0.61.1.5 192.168.145.50
gauss 0.61.1.6 192.168.145.51
```