IPACT
Queue & Routing
Services

Integrated **P**rocess **A**utomation &
Control **T**echnologies

IPACT Queue and Router Services

User Reference Manual

Document Revision: March 20, 2006

Software Version: 4.1

Operating System: OpenVMS AXP

Integrated **P**rocess **A**utomation and **C**ontrol **T**echnologies
260 South Campbell
Valparaiso, IN 46383
(219) 464-7212
Fax: (219) 462-5387

March 2006

Please notify IPACT of any errors or omissions of this document.

This document was created using Microsoft Word for Windows.

TABLE OF CONTENTS

1. Introduction

1.1 Introduction

The IPACT Queuer and Router Services (IQR) provides a standard Application Programming Interface (API) for sending messages. By using IQR, application programmers relieved of the burden of developing messaging methods between applications on the same or multiple nodes (via a network connection or through a cluster).

IQR provides delivery, recovery, and connectivity between n multiple nodes using a router installed over DECnet or TCP/IP. IQR services are provided that allow for the addition of user-supplied routers to alternate networks. IPACT has a library of other routers written for process control devices, SNA, and other networks.

A link library is provided that interfaces the Manufacturing Automation Queuing and Routing Software (available from DECUS).

The need to deliver transactions and events reliably between different computer systems has been identified for most process control computer systems. Neither TCP/IP or DECnet guarantee the delivery of messages at the application layer. The IQR router and IQR services provide this end-to-end delivery guarantee. The use of these two mechanisms provides the ability to deliver information from one computer system to another in applications where such guarantees are required (e.g. the CIM environment). Messages are not deleted or lost until the receiving process acknowledges the message. This can be thought of in a similar manner as a database “commit”.

1.2 Supported Systems

The IPACT Queuer and Router Services currently supports the AXP/Open VMS v6.1 and higher operating system.

1.3 Audience

This document is designed for the Application Programmer and the System Manager. The casual reader may choose to only read the Overview Chapter. This manual is intended for the programmer and installer of the IPACT Queuer and Router Services. This document assumes that the reader has sufficient knowledge of the VMS calling standards, system services, and typical system management skills.

1.4 Document Structure

This document contains an overview and a detailed description of the IQR product. The Overview Chapter gives a description of the product and the component parts. The API, the router, installation, and utilities are all documented in separate chapters.

2. Overview

2.1 General Information

The IQR software is designed to provide guaranteed message delivery between two different locations. This is done by creating a messaging **hub** which contains **message queues**. Each message queue contains actual **messages** to be read. Also, the IQR software provides a **router** which will move a message from one **hub** to another (even across different nodes).

2.2 IQR Components

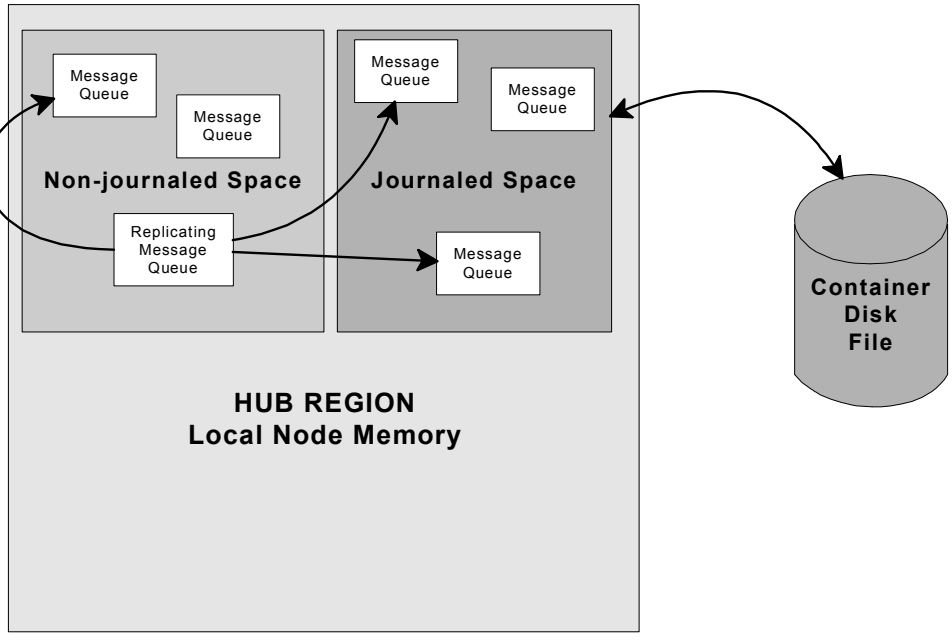
2.2.1 Hub

A hub is an individual "container" that holds all the information required for a group of message queues and their respective messages.

Each hub is created by using the IQU utility. When created, the hub occupies both system memory and a disk container file. System memory is used to store general hub information, as well as **non-journaled** message queues. The container file holds a backup of hub information and **journaled** message queues.

All hubs can contain a pre-defined number of message queues, either **journaled** or **non-journaled**. The hub is created in protected memory, so as to not allow someone to accidentally "damage" its information.

An example of a hub on a node. This shows the mapping of the Journaled space to a container disk file, and how a replicating message queue can write to messages within a hub.



2.2.2 Message Queue

Within any hub are usually many message queues. A message queue has a number of properties that all can be defined by the process creating a message queue. Message queues are created by using the IQU utility or by using the **IQR System Service**.

Messages are written to a message queue using utilities (like QIT), system service routines, or from another hub via the **router**. Messages are then read from a message queue using utilities (like DQIT), system service routines, or sent to another hub via the **router**.

All message queues are remembered between system startups. There is no need to create the queues after each startup as they will exist when the hub is re-installed using the IQU utility (see Utility Chapter). The characteristics of each message queue is also maintained over a system startup for all message queues.

The following is a list of properties that can be defined for any message queue:

Properties	***Description***
Name	All message queues have a name that consists of up to 16 characters. No two message queues in a hub can have the same name.
Size	Messages written to a message queue can be of many varying sizes. Each message queue can specify the size (in bytes) of the largest message that can be written to it.
Location	A message queue can be either **journaled** or **non-journaled**. Journaled message queues are saved in a disk file and are recoverable after abnormal events like a system crash. Additionally, journaled message queues can be shared over a cluster. Non-journaled message queues are stored in memory and offer a speed advantage over journaled message queues, but are not recoverable in the event of a system shutdown and cannot be shared over a cluster.
Volatility	Message queues can be created so that messages written to it are **volatile**. A volatile message always has the possibility of being lost in the event that the message queue has run out of room and requires more space to write a new message. In this case, the oldest message is deleted from the message queue.
Number of Messages	A message queue can be specified to hold only a certain number of messages at any one time, regardless of the size of the actual message. An example would be a volatile message queue that can hold only two messages at a time. Any time a message is written to it that would exceed the two message limit, the oldest message is deleted.

Acknowledgment — All messages must be **acknowledged** from a message queue after being read. Acknowledging a message indicates to the queue that the receiver has properly received the message, and the message queue is now clear to delete the message from the queue. Acknowledgment is usually done by the user; however, it can be set up to be done automatically after a message is read.

Number of Readers — To read a message from a message queue, you must first connect to it and declare yourself as a reader. A message queue can only have one (or two) readers connected to it at a time. The number of readers allowed is defined at the time the message queue is created. If more readers try to connect than are allowed, an error is returned. The first reader to connect is the **Primary Reader** and the second reader to connect is the **Secondary Reader**.

Stale Messages — Message queues can be set up to have **stale messages**. A message becomes stale when it exists on a queue for longer than a preset amount of time. After a message becomes stale, it is deleted from the queue (without a chance to be read).

Replication — Some message queues can be created to **replicate** any message written to it to other message queues within its hub. This can aid in the ability to perform just one write to a message queue that in turn will automatically write the message to up to four other message queues. Messages are never actually written to a replicating message queue; therefore readers are not allowed to connect to this kind of queue.

Within each message queue are its contained messages. Each message is stored in FIFO (first in, first out) order. All messages will remain in the queue until one of the following conditions is met:

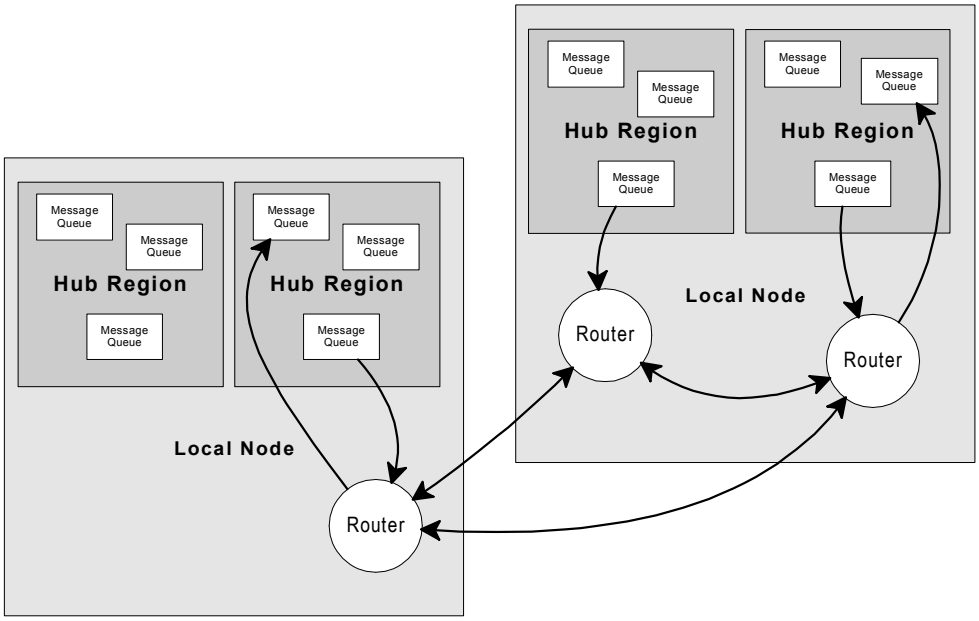
- A reader acknowledges a message
- A process requests to delete a message
- A message becomes stale
- A volatile message queue runs out of space.

Messages are not stored in any particular format. It is up to the writer/reader of the messages to interpret the actual message being passed. It is important to remember that all messages that are read must be acknowledged. If a read message is never acknowledged, the reading process will not be able to read another message until it acknowledges the current one. If, however, the process would abnormally exit and never acknowledge the message, the read message is again placed at the front of the queue. This will insure that a process will properly read each message.

2.2.3 IQR Router

The IQR Router is responsible for routing messages from message queues in a particular hub to other hubs or remote nodes. The IQR product currently supports both DECnet and TCP/IP. The remote nodes may be any DECnet, or TCP/IP, compatible node that supports the IQR Router protocol (to include routers of the MAQ/MQD and IMS type).

An example of how routers communicate. Note that a router can talk within its own node and across a network to a different router.



.

2.3 IQR System Service

The interface to the IQR software is through an **A**pplication **P**rogramming **I**nterface (API). This API is written as an user written system service and is installed with protected privileges. The system service allows the ability of the IQR software to protect the files, shared regions, and access methods from errant user programs. All of the data structures are protected in either executive or kernel mode. The following is a list of the common services provided. The IQR System Service Library Descriptions Chapter gives a more complete description of the system service calls including those typically only used by the IQR router and the IQR utilities.

- **IQR_ACK_READ** - Acknowledge a message read from a message queue
- **IQR_ATTACH_H** - Attach to a hub
- **IQR_BACKUP_RNA** - Negative acknowledgment of a message from a message queue
- **IQR_CONNECT_READ** - Connect to a message queue with intent to read
- **IQR_CONNECT_WRITE** - Connect to a message queue with intent to write
- **IQR_READ_Q** - Read a message from a message queue
- **IQR_READ_QW** - Read or wait for a message from a message queue
- **IQR_WRITE_Q** - Write a message to a message queue

2.4 Message Flow

When a VMS process writes a message to a message queue, the IQR services determine if and where the message can be queued. This determination depends on how the message queue was defined. If successful, the message will be placed in region or hub container file depending if the message queue is journaled or not. The reader process reads the message, processes the message, and then acknowledges the message from the message queue. Until the message is acknowledged the message is not deleted.

3. Installation

3.1 Command Procedure

The IQR Software is installed using the VAX/VMS INSTALL procedure in the SYS$UPDATE directory. The product name is of the form: IQR*vvr* (where *vv*=version and *r*=revision). The install kit will request a minimal number of questions to help customize your installation.

To begin your installation, insure that any previous versions of IQR are not operational. If you are installing a new version of IQR, it is recommended that you dump the messages from your message queues before beginning the installation process. To remove messages, use the following from the command line:

```
DQIT /ID=[mesg_queue] /HUB=[hub] /ALL /DUMPFILE=[filename] /ADD
```

Enter this command line for each message queue within a hub that you want to save. All messages within a message queue will be appended to the dumpfile *filename*. Note that you must have a separate filename for each hub that you want to back up. After installation, you can restore the messages by adding the message queues to the hub and then using the QIT utility to repopulate the message queues.

To begin the actual installation, enter at the command line:

```
$ @SYS$UPDATE:VMSINSTAL
```

This will begin the installation procedure. Follow the on-screen prompts and answer all questions to complete the installation. The name of the product is IQR.

Additionally, after the IQR software is installed you have the option of running the IVP (Installation Verification Procedure). This will create a test hub, create some message queues, write and read to them, and then remove the message queue. Successful operation of this process will indicate that your IQR software is now properly installed.

After the installation is completed, a file named IQR_STARTUP.COM is created in your SYS$STARTUP directory. This file must be executed before attempting to use any of the IQR System Services. The installation program will automatically execute this file; however, in order to use it after a system reset, a call to this routine should be placed into your SYSTARTUP.COM file. Use the following to execute the command procedure:

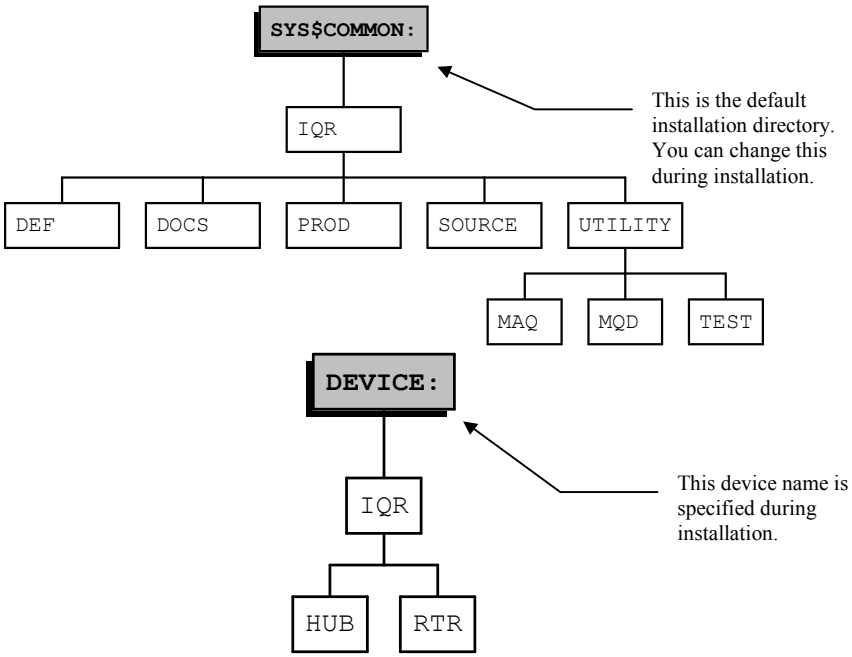
```
$ @SYS$STARTUP:IQR_STARTUP
```

Additionally, you will need to execute the following command procedure during your login to use the IQR commands. If you are getting errors regarding unknown commands, you should try executing this command procedure first:

```
$ @IQR$PROD:IQR_COMMANDS
```

3.2 Sample Directory Structure

The default directory structure created by the installation process is as follows:



Alternatively, the installation routine can install the SYS$COMMON files onto a separate device. In addition, a command procedure, IQR_STARTUP.COM, is created in your SYS$STARTUP directory.

The subroutines in each of the directories will contain the following:

Directory Name	***Contents***
DEF	Contains all of the C and FORTRAN header files to be used by a user's source code. A library is also created (in the SOURCE directory) for all of the C header files.
DOCS	This contains your online documentation and release notes.
PROD	This contains all of the executable code and the system service. The logical **IQR$PROD** is defined to this directory.
SOURCE	This contains any necessary object files, libraries, or linkable code. The logical **IQR$LIB** is defined to this directory.
HUB	The location for all hub container files. The logical **IQR$QQQQ** is defined to this directory.
RTR	The location for all router database files. The logical **IQR$RTR** is defined to this directory.
MAQ	Test utility for MAQ compatibility.
MQD	Test utility for MQD compatibility.
TEST	Test utility for IQR System Service functionality.

3.3 IQR Logicals

The following are the defined logicals used by the IQR System Service. Most of these logicals are set up by running the command procedure IQR_STARTUP.COM in the SYS$STARTUP directory.

Logical Name	***Type***	***Description***
IQRSS	**System**	This is assigned to the directory containing the IQR System Service shared system service file (IQRSS.EXE).
IQR$PROD	**System**	This is assigned to the directory containing all of the executable code for the IQR Software. This includes all utilities and command procedures.
IQR$LIB	**System**	This is assigned to the directory that contains all linkable object libraries, header libraries, text libraries, and object files.
IQR$RTR	**System/Group**	This is assigned to the directory that contains the router database files.
IQR$QQQQ	**System/Group**	This is assigned to the directory that contains the hub message queue container files.
IQR$EXAMPLES	**System**	This is assigned to the directory that contains the example programs in C and FORTRAN.
IQRHUB	**System/Group**	This is defined to be the default hub name.
RTRDEF	**System/Group**	This is defined to be the default router database name.

3.4 Command Procedures

The following command procedures are included with the installation:

Procedure Name	***Location***	***Description***
IQR_COMMANDS.COM	**IQR$PROD**	This will setup all of the IQR utility commands as foreign commands. You should execute this file during login if you intend on using the IQR utility. This may not be necessary if you installed the CLD files into your system command definition file.
IQR_STARTUP.COM	**SYS$STARTUP**	This will define all system-wide logicals necessary for the operation of the IQR System Service. In addition, it will install the System Service into memory.
***xxxx*_START.COM**	**IQR$RTR**	This will start the router named *xxxx*. This command procedure is created by the RTRDBS utility when it compiles a router database.
IQR_START_*xxxx*.COM	**IQR$PROD**	This will start your default hub (named *xxxx*) that you specified during installation. *This will not initially create the hub*. To create the hub, see the IQU /CREATE utility chapter of this manual.

3.5 Test Utilities

The installation routine will install some test utilities to test the functionality of the IQR System Service. The test utilities are provided along with their source code. To run the test utilities the user must first compile the programs with the included command procedure in their respective directories. The base location for the utilities is normally the SYS$COMMON:[IQRvvr.UTILITIES] directory. The following test routines are provided:

Test Utility	***Description***
IQR_TEST	This tests all of the basic IQR System Service calls. To compile, execute IQR_TEST.COM and then run IQR_TEST.EXE.
MQD	This tests the compatibility of the IQR System Service to the older MQD System Service. To compile, execute MENU_MQD.COM and then run MENU.EXE. You must have already installed a HUB on your system with message queues to use this program.

MAQ	This tests the compatibility of the IQR System Service to the older MAQ System Service. To compile, execute MENU_MAQ.COM and then run MENU.EXE. You must have already installed a HUB on your system to use this program.

3.6 Required Privileges

The installer should have the following privileges in order to install the IQR System Service:

- CMEXEC
- GRPNAM
- GRPPRV
- PRMGBL
- SETPRV
- SYSGBL
- SYSNAM

3.7 Sample Installation Procedure

The following is a dump from a sample installation of the IQR product. You can use it to compare against your specific installation. Items in **bold** reflect user input.

```
IPCALP$ @SYS$UPDATE:VMSINSTAL

OpenVMS AXP Software Product Installation Procedure V6.2

It is 9-AUG-1995 at 13:30.

Enter a question mark (?) at any time for help.

* Are you satisfied with the backup of your system disk [YES]? YES
* Where will the distribution volumes be mounted: DVA0

Enter the products to be processed from the first distribution volume set.
* Products: IQR
* Enter installation options you wish to use (none):

The following products will be processed:

  IQR Vxx.x

        Beginning installation of IQR Vxx.x at 13:30

%VMSINSTAL-I-RESTORE, Restoring product save set A ...


          ***********************************
          * IPACT Queuer and Router Services *
          ***********************************

                  Copyright (C) 1995 by:

                        IPACT Inc.
                   260 South Campbell
                  Valparaiso, IN  46383
          (219) 464-7212     fax (219) 462-5387

                  All rights reserved.

Your IQR Serial number: xxx-xxxxxx-xxxx

About to begin installation of the IQR Services. If
you do not want to install at this time, please
enter a N at the prompt.  Otherwise, press RETURN
and answer all questions presented.

* Are you ready to begin installation? [Y]? Y

***************** SELECT DEFAULT PATHS *****************

The IQR System Service, all source code, all executables,
```

```
all utilities, and the development environment are all
placed in the SYS$COMMON:[IQRxxx] directory.

Alternatively, any drive can be selected.  Common service
files will be placed in <drive>:[IQRxxx]. The drive
selected should be one that is mounted upon system
startup such that the system service can be installed.

* Place IQR Service in SYS$COMMON: directory? [Y]? Y

********************** SELECT FACILITY CODE ************************

The facility code for the system service and the routing process
error messages are selected by the following question.  The user
should be aware that the message codes are in the range of 1 to 2047.
The actual value is supposed to be assigned by digital, but since
this code does not come from digital, we are not able to or desire
to go through the hassle.  Instead, we will let you select the
factility number.  One should be aware, as should all users of the
queue service (not really a problem with the router), what factility
number was chosen.  If a user selects the same number, then the
translation of error messages can be all messed up (ie: QUESUC may
translate to a user defined facility code that means: BADINPUT). The
normal standard defined of facility codes for digital products can be
found in the QUEMSG macro.

The system manager may choose to add the error codes for queue
services for all users or for an individual user.  The following
command will define all of the queue service error codes.

        SET MESSAGE IQR$LIB:IQR_MSG.EXE

* Enter the QUEUE SERVICE facility number (1 to 2047) [9]: 9

*************** SELECT CONTAINER DEVICE ***************

The IQR System Service stores all HUB and ROUTER database
information on any mounted disk.  Note that the HUB
container file is implemented as an RMS paged file that
stores your entire hub information and all journaled
messages.

Ideally, this should NOT be your system disk.  This disk
should be on a disk that is not heavily used or at least
some thought should be given to partitioning loads for
the I/O required to checkpoint the global section to the
RMS page file.  Also, a disk with a high access and read
speeds will greatly improve the speed of the IQR services
that reference the HUB container file.

This disk will need to be mounted before any IQR operations
can be performed (ideally in your system startup).

* Which device should contain Queue container file: DKA500

The hub file itself can be any size but will require system
global pages and disk space.  You should use the INSTALL
utility to determine how many free pages are available, and
add the number you intend to allocate for the Queue container
file.   Finally, each process that connects to the global
section must be able to map the queue container file,
therefore, the virtual page count must be large enough.  For
this software 3000 plus the number of pages allocated for the
queue file is adequate. A normal size for the queue file is
2000 to 5000 pages.

Below are the current settings of these sysgen parameters.

Number pages allocated for global sections: 166208
Current number of global pages free: 48576
System Process largest Virtual Address Space: 139264

* Do you wish to change these and reinstall this kit later [N]? N

***************** DEFAULT HUB NAME *****************

The installation routine will set up a file that will
initalize a default HUB for your message queues.  This
must be no more than 8 characters and have no blanks.
The file will be called IQR_START_<name>.COM and should
be executed in order to install the hub on the system.
Note that you may have to first create the hub on the
system before you can install it.  See the user's
manual on the IQU utility for more information.

* Enter the name of default message hub file: ALPHA

***************** COMMAND DEFINITON *****************

You have the option of installing the IQR commands into
your system CLD tables.  If you do this, then anyone
who logs in with a copy of these tables will get access
to the IQR commands.  If you do not choose this option,
then you will have to run the following command
procedure in order to set up the IQR utilities as
foreign commands:

        @IQR$PROD:IQR_COMMANDS

This is the recommended procedure which will keep
```

```
things from getting messed up in the event the
system CLD tables are replaced.

* Install CLD files? [N]? N

****************** END Q/A SESSION ******************

* Are you happy with your answers [Y]? Y

*************** CONTINUING INSTALLATION ***************

%IQR-I-MILLERTIME, Interaction section is complete - installation continuing

 Creating common directory tree
 Resulting Directory Tree for this product will be:

        SYS$COMMON:[IQR004]
                      +-[DEF]
                      +-[DOCS]
                      +-[PROD]
                      +-[SOURCE]
                      +-[UTILITY]-+
                                  |
                                  +-[TEST]
                                  +-[MAQ]
                                  +-[MQD]
        _IPCALP$DKA500:[IQR004]
                 +-[HUB]
                 +-[RTR]

Creating directory tree
%VMSINSTAL-I-SYSDIR, This product creates system disk directory  IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004].
%CREATE-I-EXISTS, _IPCALP$DKA500:[IQR004] already exists
Creating definitions directory
%VMSINSTAL-I-SYSDIR, This product creates system disk directory  IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004.DEF].
Creating documentation directory
%VMSINSTAL-I-SYSDIR, This product creates system disk directory  IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004.DOCS].
Creating production directory
%VMSINSTAL-I-SYSDIR, This product creates system disk directory  IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004.PROD].
Creating source files directory
%VMSINSTAL-I-SYSDIR, This product creates system disk directory  IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004.SOURCE].
Creating utility directory
%VMSINSTAL-I-SYSDIR, This product creates system disk directory  IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004.UTILITY].
Creating utility:test directory
%VMSINSTAL-I-SYSDIR, This product creates system disk directory
IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004.UTILITY.TEST].
Creating utility:maq directory
%VMSINSTAL-I-SYSDIR, This product creates system disk directory  IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004.UTILITY.MAQ].
Creating utility:mqd directory
%VMSINSTAL-I-SYSDIR, This product creates system disk directory  IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004.UTILITY.MQD].
Creating hub container storage directory
Creating route directory
Loading documentation directory
Loading source directory
Loading default queue directory
Loading default router directory
Loading MQD test utilities
Loading MAQ test utilities
Loading IQR test utilities
Loading definitions directory
Loading production directory
Building IQR_MSG
Linking IQR_MSG and replacing in HUB Library
Linking IQR System Service
Linking utility: DMPQUE
Linking utility: DMPRTR
Linking utility: DQIT
Linking utility: QIT
Linking utility: IQU
Linking utility: LSTRTR
Linking utility: RTRDBS
Linking utility: IQR_RTR
Creating IQR startup procedure

************ INSTALLATION VERIFICATION PROCEDURE ************

This kit is supplied with an IVP that is part of
the VMSINSTAL kit.  If you choose to execute the IVP, then
the IQRSS system service will be installed.  A test hub named
TEST_HUB will then be created.  Two message queues will be
created, TESTMID1 and TESTMID2.  Some messages will then be
written to the queues.  Both message queues should only contain
two messages after writing.  Then, the queues are displayed and
then read off of the message queue.  Finnaly, the test hub is
removed from memory (and disk) and an indication of your
installation configuration is displayed.

* Execute IVP [Y]? Y

************************* FINISHING UP *************************

Your installation is now complete.  After the files are moved,
we will test your installation if you requested it.

Please remember to read the user's manual for more information
about configuring the installation for your particular needs.

%VMSINSTAL-I-MOVEFILES, Files will now be moved to their target directories...
 Installing IQRSS and defining logicals
```

```
%INSTALL-W-NOPREV, no previous entry exists - new entry created for
IPCALP$DKA300:[SYS0.SYSCOMMON.IQR004.PROD]IQRSS.EXE;1

++++ IQR installation verification procedure ++++

Defining IQR symbols
Creating TEST_HUB test container
Creating message queues

Writing messages to message queues
 Writing to message queue: TESTMID1
 Writing to message queue: TESTMID1
 Writing to message queue: TESTMID1
 Writing to message queue: TESTMID1
 Writing to message queue: TESTMID2
 Writing to message queue: TESTMID2

HUB information for: TEST_HUB on IPCALP::
HUB Operational since  9-AUG-1995 13:32:42.79  Up for    0 00:00:01.48

Location     Size    Free Blk  Write Cntr  Read Cntr  Act Queues
---------  --------  --------  ----------  ---------  ----------
Container     12104     12096           0          0           0
Region          197       120           6          0           2

Queue Name      Flags   CurMsg MaxMsg Last Wrt Last Ack CumTran
-------------- -------- ------ ------ -------- -------- -------
TESTMID2       ........      2     20 13:32:44 00:00:00 0:00:00
TESTMID1       .......V      2      2 13:32:43 13:32:43 0:00:00

 Read from message queue: TESTMID1
 Header follows:
     Source node name:
     Destination node name:
     Message type:                0
     Sequence number:             3
     Message length:             22
     On Queue time:           9-AUG-1995 13:32:43.74
 Total on queue time:             0days and 00:00:00.78
 ------- MESSAGE FOLLOWS ---------
 this is test message3

 Header follows:
     Source node name:
     Destination node name:
     Message type:                0
     Sequence number:             4
     Message length:             22
     On Queue time:           9-AUG-1995 13:32:43.86
 Total on queue time:             0days and 00:00:00.89
 ------- MESSAGE FOLLOWS ---------
 this is test message4
 End of message queue reached.
%IQRSRV-W-NOMESS, No message for the specified message queue

 Read from message queue: TESTMID2
 Header follows:
     Source node name:
     Destination node name:
     Message type:                0
     Sequence number:             1
     Message length:             22
     On Queue time:           9-AUG-1995 13:32:43.97
 Total on queue time:             0days and 00:00:00.95
 ------- MESSAGE FOLLOWS ---------
 this is test message1

 Header follows:
     Source node name:
     Destination node name:
     Message type:                0
     Sequence number:             2
     Message length:             22
     On Queue time:           9-AUG-1995 13:32:44.09
 Total on queue time:             0days and 00:00:00.99
 ------- MESSAGE FOLLOWS ---------
 this is test message2
 End of message queue reached.
%IQRSRV-W-NOMESS, No message for the specified message queue

 Removing TEST_HUB from memory and disk

IQR -- IPACT Queuer and Router

Serial #xxx-xxxxxx-xxxx
  Version : xx
  Revision: x

 **** Verification procedure complete ****

        Installation of IQR Vxx.x completed at 13:32

    Adding history entry in VMI$ROOT:[SYSUPD]VMSINSTAL.HISTORY
    Creating installation data file: VMI$ROOT:[SYSUPD]IQR004.VMI_DATA

Enter the products to be processed from the next distribution volume set.
* Products:

        VMSINSTAL procedure done at 13:33
```

4. IQR System Service Library

4.1 iqr_ack_read

Acknowledge an already read message from a message queue.

FORMAT

iqr_ack_read (hub, queue_name, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of acknowledgment of message from the message queue.

ARGUMENTS

hub

Type:	Record of type hbkdef
Access:	Read only
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

Message queue name to acknowledge.

queue_index

Type:	Longword
Access:	Read only
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This routine will acknowledge a message that has been read but not yet acknowledged on a particular message queue. For any message, once it has been read, it will need to be acknowledged from the queue to re-allocate space in the queue. An acknowledgment will indicate that the calling process has properly received the message and that the queue may delete the message and prepare to read the next.

Also, further messages will be unable to be read until the last read message is acknowledged. Message queues that are set up with the MQD_M_ACKREAD flag set will automatically acknowledge the message from the message queue once it is read via **iqr_read_q** or **iqr_read_qw**.

Failure to acknowledge a message from the queue and then disconnecting will result in a stale message being left on the message queue. When a reader reconnects to the message queue, the stale message will be placed back at the head of the message queue.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully acknowledge the message off of the queue.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_NOTCONREAD	The caller has not yet connected for read to this queue.
QUE_NORNAMESS	No message was read off of the queue -- the RNA has not been set. You need to first read a message before you can acknowledge it.

4.2 iqr_add_message_q

Creates a new message queue (journaled or non-journaled) within a specific hub.

FORMAT

iqr_add_message_q (hub, queue_name, max_messages, stale_time, type, max_mesg_size)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of queue creation including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read only
Mechanism:	By reference

Hub is a buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Character descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the queue to be created in descriptor format - maximum of 16 characters.

max_messages

Type:	Longword
Access:	Read only
Mechanism:	By value

This will indicate the maximum number of queued messages that will be allowed on the queue at any one time. This does *not* include messages currently read but not yet acknowledged.

stale_time

Type:	Longword
Access:	Read only
Mechanism:	By value

This value will reference the number of minutes that messages on the queue will remain as valid messages. After that time, messages will become *stale* and deleted from the queue. Requires the value MQD_M_TIMED to be set for **type**.

type

Type:	Longword
Access:	Read only
Mechanism:	By value

This is set to indicate the type of message queue to be created. Valid types are as follows:

MQD_M_ACKREAD	Automatically acknowledge a message when it is read.
MQD_M_DUALREAD	Message queue supports both a primary and secondary reader (two readers).
MQD_M_JOURNALED	Messages are kept in journaled space (on disk) instead of within memory.
MQD_M_READER	Do not queue (write) messages unless there is a reader connected to the message queue.
MQD_M_REPLICATING	Writing to this message queue will instead queue the message to multiple message queues on the current hub (replicate).
MQD_M_TIMED	Timed message queue. Stale messages are removed from the queue. Requires a value for **stale_time**.
MQD_M_VOLATILE	Message queue can contain volatile messages. The oldest message will be deleted if there is not room for a new one.

max_mesg_size

Type:	Longword
Access:	Read only
Mechanism:	By value

This is set to the size (in bytes) of the largest message that will be allowed to be written to the queue.

DESCRIPTION

This routine is used to create or add a new message queue to an existing hub. In order to be able to read/write messages to a queue, the queue must first be created with **iqr_add_message_q**.

If the message queue already exists within the hub (and has not been deleted), an error message is returned.

The routine will build the message queue in memory (or on disk if the MQD_M_JOURNALED option is specified) and reset all message queue counters.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully created the message queue.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hbk was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_CONTAINERFULL	The disk container is at its maximum size.
QUE_NOCACHE	No cache space available for the message queue.
QUE_ADDED	The queue was successfully added.

4.3 iqr_allocate_msgblks

Allocate space in a message queue for a multi-packet message.

FORMAT

iqr_allocate_msgblks (hub, queue_name, msg_size, hmb_blck, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of allocation, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to write to.

msg_size

Type:	Longword
Access:	Read only
Mechanism:	By value

This is the size of the message to be written. This *does not* include the message's header.

hmb_blck

Type:	Longword
Access:	Write
Mechanism:	By reference

This is the returned value of the block number for the header message block created by the routine. This is used by other calls to **iqr_fill_msgblks** and **iqr_thread_msgblks**.

queue_index

Type:	Longword
Access:	Read
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This service entry is called by a router or another program that desires to allocate space for a message that will later be populated and then threaded into the message queue chain. If the allocated blocks are not needed after they are allocated, then the service **iqr_deallocate_msgblks** may be used to return them back to the free space.

This routine is used in conjunction with the routines **iqr_fill_msgblks** and **iqr_thread_msgblks** to first populate and then thread the message into the queue.

This service requires that the user has the SYSNAME privilege. This test is made to ensure that the caller is a more knowledgeable user. The privilege is not actually needed by the service.

The service functions somewhat like the **iqr_write_q** service by checking to make sure that the message queue will accept another message and that the user is correctly connected to the message queue before actually doing anything to the queue.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successful completion.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.

QUE_NOTCONWRITE	The caller has not yet connected for write (or read) to this queue.
QUE_MQDFULL	The message queue is full (the number of messages in the queue exceeds that set by iqr_add_message_q).

4.4 iqr_attach_h

Attach to a messaging hub.

FORMAT

iqr_attach_h (hub [, hub_name] [, mqd_count])

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of hub attachment including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read/Write
Mechanism:	By reference

This is the returned hub reference required for any calls to the IQR service routines.

hub_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

Contains the name of the hub to map, maximum of 8 characters. *(Optional. Defaults to Hub defined by logical IQRHUB)*

mqd_count

Type:	Longword
Access:	Read only
Mechanism:	By value

This specifies the maximum number of message queues that the caller intends to connect to. *(Optional. Defaults to an internal value of 4)*

DESCRIPTION

This routine creates an area for the process (PEX - process expanded region) where the process can store information on the queue process and provide working space for other queue services.

The starting and ending address of the PEX is returned to the caller in the *hub* parameter.

Additionally, the routine establishes an exit handler for the process which will disconnect from all queues and the hub upon exit.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully attached to the message hub.
QUE_PREATT	Informational: indicates that the message hub has already been attached.
QUE_INVALIDPEX	User passed hbk was invalid. Missing or not writeable. Usually caused by not having the hbk initialized through the **iqr_attach_h** routine.
QUE_BADHNAME	User passed hub name was invalid. Name was too short, too long, or was not accessible for read.
QUE_DEFHNAME	Unable to translate the default hub name. Most likely logical not defined by the system manager.
QUE_BADCCTMQD	The user specified number of message queues exceeds a reasonability test defined by the software. Contact developers if this is too low for your environment. The maximum number was specified when the software was packaged.
QUE_BADPRCINF	The queue service was unable to get information about the current process. Examine the condition code in the extended status field of the hbk. Contact the developers.

QUE_PRCLCKNM	The queue service was unable to create a process lock. Examine the condition code in the extended status field of the hbk.
QUE_BADSIG	The queue service was unable to establish a signal resource lock. Examine the condition code in the extended status field of the hbk.
QUE_BADPRCLCK	Unable to capture process lock. Examine the condition code in the extended status field of the hbk.
QUE_BADSRV	Internal error. Unable to demote lock on the signal resource.

Additionally, extended status is provided in the hub structure. If the error is QUE_ACCVIO, the number of the parameter that was not readable or writeable is stored in this entry if the hub was at least writeable.

4.5 iqr_backup_rna

Restore a read, but not yet acknowledged message back onto the front of a message queue.

FORMAT

iqr_backup_rna (hub, queue_name, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of restore, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to backup the message.

queue_index

Type:	Longword
Access:	Read
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This service is called to backup a message that has been read but not yet acknowledged to the front of the message queue. After reading a message from a message queue, the message will remain "held" until the message is acknowledged off of the queue. If the reader desires to return the message back to queue, then call this routine *before* acknowledging the message. All backed-up messages are returned to the *front* of the message queue.

This action is usually done during abnormal process rundown, but the user may also desire a way to easily return a message back onto the queue to be read again later.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully restored the message back onto the message queue.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_NOTCONREAD	The caller has not yet connected for read to this queue.
QUE_NORNAMESS	No message was read off of the queue -- the RNA has not been set. You need to first read a message before you can acknowledge it.

4.6 iqr_connect_read

Connect to a message queue for subsequent reading.

FORMAT

iqr_connect_read (hbk, queue_name, event_sync, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of message queue connection, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

Hub is a buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

Message queue name, 16 characters max.

event_sync

Type:	Longword
Access:	Read only
Mechanism:	By value

Event flag for synchronization.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By reference

This is a pointer to a longword that will hold the memory address for the Connected Message Queue (CMQ) definition created by the connection. This value will be required by any subsequent calls that use this particular message queue.

DESCRIPTION

This service defines the calling process as a reader of a particular message queue. In order to connect for read, the user must have already attached to the message hub and the message queue must already exist.

If these conditions are met, a new connect message queue block is allocated in the user's process expanded region. Then an attempt is made to capture the message queue lock. If the lock can be captured in exclusive mode, then the caller is the one and only primary reader of the message queue. If the lock cannot be captured, then another user is currently connected for read and an error is returned. Also, an AST is created by the service which will set the caller's event flag when one of the IQR Service routines wishes to notify all readers of this message queue that a new message has arrived.

If the message queue was created with the MQD_M_DUALREAD option, then a secondary reader can also be connected as well. At any one time, only ***one*** primary and ***one*** secondary reader can be connected. Any other attempts to connect as a reader will result in the error QUE_TOOMANYRDR.

Upon connection the message queue is checked for messages that have been read but not yet acknowledged. If the reader who read the message is no longer connected to the queue, then that message is again re-queued at the beginning of the queue.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully attached to the message queue for read within the hub.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length. Can also indicate trying to connect to a replicating queue (which is invalid).
QUE_INVARG	User passed argument invalid, or not readable.
QUE_MAXMSGQUEUES	User has exceeded the maximum number of message queues that may be attached. The number specified on the **iqr_attach_h** or **mqd_count** must be increased.
QUE_TOOMANYRDR	Maximum number of readers already connected to the message queue.
QUE_INVALIDPEX	User passed hbk was invalid. Missing or not writeable.

QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.

4.7 iqr_connect_write

Connect to a message queue for subsequent writing.

FORMAT

iqr_connect_write (hub, queue_name, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of connection, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to connect to, maximum of 16 characters.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By reference

This is a pointer to a longword that will hold the memory address for the Connected Message Queue (CMQ) definition created by the connection. This value will be required by any subsequent calls that use this particular message queue.

DESCRIPTION

This routine defines the caller as a writer to a particular message queue. In order to connect for write, the user must have already attached to the message hub and the message queue must already exist.

If the conditions are met, then the caller is connected for write to the message queue. Any number of writers can be connected to the queue at any one time.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully attached to the message queue for write within the hub.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_MAXMSGQUEUES	User has exceeded the maximum number of message queues that may be attached. The number specified on the **iqr_attach_h** or **mqd_count** must be increased.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.

4.8 iqr_deallocate_msgblks

This will release allocated blocks (from **iqr_allocate_msgblks**) back to the queue.

FORMAT

iqr_deallocate_msgblks (hub, queue_name, hmb_blck, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of deallocation, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to use, 16 characters maximum.

hmb_blck

Type:	Longword
Access:	Read
Mechanism:	By value

This is a pointer to the value of the block number of the header message block for the message that is to be deallocated.

queue_index

Type:	Longword
Access:	Read
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This routine will release blocks that have been allocated to a message and its header for a particular message queue.

These blocks must have been allocated with the **iqr_allocate_msgblks** service. Generally, this routine is used as a rundown or error condition service to free allocated blocks back to the message queue if the message could not be properly written.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successful return.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_NOTCONWRITE	The caller has not yet connected for write (or read) to this queue.

4.9 iqr_delete_q

This routine will mark a message queue as deleted from a specific hub.

FORMAT

iqr_delete_q (hub, queue_name)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of queue deletion, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read only
Mechanism:	By reference

Hub is a buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Character descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the queue to be deleted in descriptor format - maximum of 16 characters.

DESCRIPTION

This routine will search for the passed message queue name within the indicated hub. If it is found, then it will mark that queue as being deleted. Any future attempts to read/write to the queue will not be allowed. The message queue must not contain any waiting messages in order to be deleted.

Note that the information for the queue is not actually deleted -- it is only marked as invalid. This can allow for the re-creation of the queue in the event it is needed.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successful modification of the message queue.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable. Also can indicate that the message queue still has waiting messages.
QUE_INVALIDPEX	User passed hbk was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.

4.10 iqr_disconnect_h

Disconnect the calling process from all connected hubs and message queues (rundown service).

FORMAT

iqr_disconnect_h (hub)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of disconnection, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

DESCRIPTION

This service is used as a part of the general rundown for the IQR System Service. It will disconnect the caller from all connected message queues, zeroing out the caller's process expanded region. Also, it will remove all process locks created by the IQR Service calls.

This routine should be called upon completion of the caller's program to insure that all locks are released and resources are returned to the system.

Additionally, this service is called as a part of the CTRL-Y AST handler in order to properly allow for a rundown of the IQR services.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully completed.

4.11 iqr_disconnect_q

This service will allow the caller to disconnect from a message queue.

FORMAT

iqr_disconnect_q (hub, queue_name, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of disconnect, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to disconnect.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This service will disconnect the caller from a particular message queue. If the caller has connected for read or write, this routine will disconnect him from the message queue and mark his connection block as invalid.

To access a message queue again, you must reconnect using **iqr_connect_read** or **iqr_connect_write** service.

You should call this service when you have finished working with a message queue. This will insure that all information is properly handled within the queue, and in the case of readers, will open up the availability for another process to connect as a reader.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully disconnected from the message queue.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hbk was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_INVALQIDX	The queue_index is not valid.

4.12 iqr_fill_msgblks

Fill previously allocated message blocks in a message queue.

FORMAT

iqr_write_q (hub, queue_name, size, offset, buffer, hmb_blck, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of fill, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to write to.

size

Type:	Longword
Access:	Read only
Mechanism:	By value

This is the size (in bytes) of the current segment to be written to the queue. This *is not* the total size of the message, but just the size for the current packet. Unless this is the last segment to be written, the value must be a multiple of 512 bytes.

offset

Type:	Longword
Access:	Read only
Mechanism:	By value

The current byte offset of the current packet that is being written to a message queue. The offset is relative to the start of the message and should always be a multiple of 512 bytes.

buffer

Type:	Longword block index
Access:	Read
Mechanism:	By reference

This is a pointer to a block buffer where the information is to be placed on the queue. This *does not* include a message header.

hmb_blck

Type:	Longword
Access:	Read
Mechanism:	By value

This is the value of the block number of the header message block for the message that is to be threaded.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By reference

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This service is designed to allow a privileged user the ability to populate the blocks of a message that was previously allocated using the **iqr_allocate_msgblks**. Since there exists the ability to corrupt the threads in the hub, this service will test to ensure that the caller has the SYSNAM privilege.

This service is primarily used to fill allocated message space with multiple "packets". Each packet is a small part of the entire message, and when completed, will fill the blocks in the queue identical to that of a basic **iqr_write_q** service. The parameters **size** and **offset** are used to keep track of the current position being written in the current message. **Size** must always be a multiple of 512 bytes (unless you are writing the final packet, in which **size** may be smaller than 512 bytes). **Offset** is a user-maintained value that tells the service where in the message you are currently writing. **Offset** is calculated from the beginning of the actual message, disregarding any headers, and must always be a multiple of 512 bytes.

Before writing any data, the hmb block is checked to make sure it is valid. The *hmb_blck* value passed must match that which was stored there by the **iqr_allocate_msgblks** service.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successful completion.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_INVALUSERBUF	The buffer passed by the user is invalid
QUE_NOTCONWRITE	The caller has not yet connected for write (or read) to this queue.
QUE_MQDFULL	The message queue is full (the number of messages in the queue exceeds that set by iqr_add_message_q).

4.13 iqr_get_q_info

Gather information about a specific message queue.

FORMAT

iqr_get_q_info (hub, queue_name, q_info)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of message queue connection, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

Hub is a buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

Message queue name, 16 characters max.

q_info

Type:	Record structure MQDDEF
Access:	Write
Mechanism:	by reference

This is a pointer to a buffer with a structure type of MQDDEF (found in MQDDEF.H) to hold the message queue information.

DESCRIPTION

This routine will copy all of a message queue's information into the passed structure pointed to by ***q_info***.

Some notable information presented in the message queue are:

q_info->mqd._q.L_type	Type of message queue
q_info->mqd._q.Lcnt	Number of messages in the queue
q_info->mqd._q.L_rna	Last read, but not yet acknowledged message id

q_info->mqd._q.L_srna	Last read, but not yet acknowledged message id via secondary reader.
q_info->mqd._q.max_cnt	Maximum number of messages allowed on a message queue.
q_info->mqd._q.expire	Time (in minutes) before a message becomes stale.
q_info->mqd._q.msg_name	Name of the message queue
q_info->mqd._q.L_lost	Number of messages lost due to full message queue.

Note that the information returned represents just a *copy* of the current state of the message queue. Therefore, the actual state of the queue may change after you have received your information.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully attached to the message queue for write within the hub.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.

4.14 iqr_modify_q

Modifies a specific message queue's parameters including message queue type flags, time for stale messages, maximum message size, and maximum number of messages in the queue at one time.

FORMAT

iqr_modify_q (hub, queue_name, max_mesg, stale_time, queue_type, max_msgsize)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of queue modification, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read only
Mechanism:	By reference

Hub is a buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Character descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the queue to be modified in descriptor format - maximum of 16 characters.

max_messages

Type:	Longword
Access:	Read only
Mechanism:	By value

This will indicate the maximum number of queued messages that will be allowed on the queue at any one time. Set this value to zero if you do not wish to modify this parameter.

stale_time

Type:	Longword
Access:	Read only
Mechanism:	By value

This value will reference the number of minutes that messages on the queue will remain as valid messages. After that time, messages will become *stale* and deleted from within the queue. Requires the value MQD_M_TIMED to be set for **type**. Set this value to zero if you do not wish to modify this parameter.

type

Type:	Longword
Access:	Read only
Mechanism:	By value

Set this argument to any of the following codes to modify the message queue's parameters. If you do not specify a code it will then be cleared in the message queue. Logical OR the following codes to select more than one option. Valid types are as follows:

MQD_M_ACKREAD	Automatically acknowledge a message when it is read.
MQD_M_DUALREAD	Message queue supports both a primary and secondary reader (two readers).
MQD_M_READER	Do not queue (write) messages unless there is a reader connected to the message queue.
MQD_M_TIMED	Timed message queue. Stale messages are removed from the queue. Requires a value for **stale_time**.
MQD_M_VOLATILE	Message queue can contain volatile messages. The oldest message will be deleted if there is not room for a new one.
MQD_M_NOCHANGE	Use this value if you **do not** want to change any of the current values for the message queue flags.

max_mesg_size

Type:	Longword
Access:	Read only
Mechanism:	By value

This is set to the size (in bytes) of the largest message that will be allowed to be written to the queue. Set this value to zero if you do not wish to modify this parameter.

DESCRIPTION

This routine is used to modify an existing message queue on an existing hub region. Use this routine if you wish to modify one of the passed parameters for the message queue.

NOTE: If you ***do not*** want to change the flags for the queue, ***you must*** pass the parameter MQD_Q_NOCHANGE. Failure to do so will result in all of the flags being reset to a value of zero.

If you ***do not*** want to change any of the other parameters (stale time, maximum message size, etc.), set them to a value of zero.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successful modification of the message queue.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hbk was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.

4.15 iqr_read_hmb

This service is used to only read the header of the next available message on a message queue.

FORMAT

iqr_read_hmb (hub, queue_name, user_header, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of read, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to read from.

user_header

Type:	Record structure hdrdef
Access:	Write
Mechanism:	By reference

This will hold the header information for the current message being read from the queue.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This service will read the next message's header information from the queue. All queue information is preserved, including the current message on the queue.

This routine is used to let the calling program determine the characteristics of the next message to be read on the queue (such as message size, time written, etc.) without actually reading the message off of the queue.

It is possible, with dual readers, that the message read from **iqr_read_hmb** will *not* be the same one read by a successive read call. This can happen if the other reader reads the message before the calling process actually gets to read the message. In order to avoid this conflict, it is recommended that message queues for processes that require using this routine (including the IQR Router) be set up to *not use dual readers*.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successful read of header from the queue.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_NOTCONREAD	The caller has not yet connected for read to this queue.
QUE_USRBUFSML	The size of the passed buffer is too small.
QUE_NOMESS	The message queue is currently empty. No messages exist to read.

4.16 iqr_read_q

Read the next message from a message queue.

FORMAT

iqr_read_q (hub, queue_name, buffer, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of read, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to read from.

buffer

Type:	Descriptor
Access:	Write
Mechanism:	By reference

This is a descriptor for the buffer where the read information is to be written. This buffer ***must*** be large enough to hold ***both*** the message header block (HMBDEF) and the message itself.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This routine will attempt to read the next message from the given message queue.

If a message exists, then the message's header information and the message itself are returned in the passed buffer location. The actual message will be located at an offset of size HBKDEF (or HDR$K_SIZ) from the start of the buffer.

If the message queue is empty, then a condition of QUE_NOMESS is a returned indication that there are no messages currently in the queue.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully read a message.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_INVALUSERBUF	The buffer passed by the user is invalid.
QUE_NOTCONREAD	The caller has not yet connected for read to this queue.
QUE_USRBUFSML	The size of the passed buffer is too small.
QUE_NOMESS	The message queue is currently empty. No messages exist to read.

4.17 iqr_read_qn

Read the next message from a message queue
or if empty, trigger user AST when new
message arrives.

FORMAT

iqr_read_qn (hub, queue_name, buffer, queue_index,ast,astprm)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of read, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to read from.

buffer

Type:	Descriptor
Access:	Write
Mechanism:	By reference

This is a descriptor for the buffer where the read information is to be written. This buffer ***must*** be large enough to hold ***both*** the message header block (HMBDEF) and the message itself.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

ast

Type:	Longword
Access:	Write
Mechanism:	By value

This is the user mode AST procedure to be called.

Astprm

Type:	Longword
Access:	Write
Mechanism:	By value

This is the parameter that is passed to the user procedure.

DESCRIPTION

This routine will attempt to read the next message from the given message queue.

If a message exists, then the message's header information and the message itself are returned in the passed buffer location. The actual message will be located at an offset of size HBKDEF (or HDR$K_SIZ) from the start of the buffer.

If the message queue is empty, then a condition of QUE_NOMESS is a returned indication that there are no messages currently in the queue and the queue is marked to receive a notification when a writer completes a write to the queue. It should be noted that the notification does not always guarantee a message has been placed into the queue as there are conditions which can result in a false trigger. A typical FORTRAN usage is shown below (only portions of the code actually shown):

```
Program main
integer*4 Notify_ast
external Notify_ast
STATUS=IQR_READ_QN(HUB,ID,d_buffer,%val(index),notify_ast,21)
end

Subroutine notify_ast(p_ast,r0,r1,psl,pc)
integer*4 p_ast,r0,r1,psl,pc
integer*4 astparm_ast
call sys$setef(%val(1))
return
end
```

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully read a message.

QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable (e.g., AST parameter is null)
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_INVALUSERBUF	The buffer passed by the user is invalid.
QUE_NOTCONREAD	The caller has not yet connected for read to this queue.
QUE_USRBUFSML	The size of the passed buffer is too small.
QUE_NOMESS	The message queue is currently empty. No messages exist to read.

4.18 iqr_read_qw

Read the next message from a message queue if one does not exist, wait for one to arrive.

FORMAT

iqr_read_qw (hub, queue_name, buffer, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of read, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to read from.

buffer

Type:	Descriptor
Access:	Write
Mechanism:	By reference

This is a descriptor for the buffer where the read information is to be written. This buffer ***must*** be large enough to hold ***both*** the message header block (HMBDEF) and the message itself.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This routine is very much like **iqr_read_q** except that it will wait for a message to arrive if none currently exists in the queue.

If a message exists, then the message's header information and the message itself are returned in the passed buffer location. The actual message will be located at an offset of size HDR$K_SIZ from the start of the buffer.

If the message queue is empty, then the routine will wait for a message to arrive at the queue. After the message arrives, it will then repeat the process of reading the message.

Note that with ***two*** readers (primary and secondary) only one of the readers will be able to get a message when it first arrives to an empty queue. The one that is unable to get the message will again go into a wait mode unless more messages exist in the queue.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully read a message from the queue.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_INVALUSERBUF	The buffer passed by the user is invalid.
QUE_NOTCONREAD	The caller has not yet connected for read to this queue.
QUE_USRBUFSML	The size of the passed buffer is too small.

4.19 iqr_read_segment

This service is used to read messages off of the queue in individual segments at a time.

FORMAT

iqr_read_segment (hub, user_header, buffer_size, buffer, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of read, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

user_header

Type:	Record structure hdrdef
Access:	Write
Mechanism:	By reference

This will hold the header information for the current message being read from the queue.

buffer_size

Type:	Longword
Access:	Read
Mechanism:	By value

This is the size (in bytes) of the read buffer. The size must be a multiple of 512 bytes (one block)

buffer

Type:	Descriptor
Access:	Write
Mechanism:	By reference

This is a descriptor for the buffer where the message segment is to be written. The buffer will contain only the current message segment -- no headers.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This service will read messages off of the queue in individual segments at a time. This routine will keep track of where it is currently reading from in the message.

Upon the initial read of a message, the service will return the header information of the message along with the first message segment. Further calls to the service will return successive segments of the message with each call (along with the header). Upon reading the last segment of the message, the routine will return a status code of QUE_LASTSEG, signaling that the end of the message has been reached.

You may acknowledge the message off of the queue at any time during the read of the segments, but will not be able to read a new message until the current one is acknowledged. In order to insure that you do not prematurely acknowledge a message, wait until this service returns a status code of QUE_LASTSEG before acknowledging.

Normally, this routine is used by the router reading segments of a message and then sending the segments to another router.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_LASTSEG	Successfully read segment from the queue (no more segments exist, must acknowledge message before reading again).
QUE_SUCCESS	Successfully read segment from the queue (more segments still exist for message).
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.

QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_INVALUSERBUF	The buffer passed by the user is invalid.
QUE_NOTCONREAD	The caller has not yet connected for read to this queue.
QUE_USRBUFSML	The size of the passed buffer is too small.
QUE_NOMESS	The message queue is currently empty. No messages exist to read.

4.20 iqr_reset_stat_h

This service is used to reset statistical counters and timers for a message hub.

FORMAT

iqr_reset_stat_h (hub)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of read, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

DESCRIPTION

This service will reset statistics for a message hub. All timers, except for the time of modification, will be reset to a time of zero. In addition, the number of reads and writes for the message queues will be reset to zero.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully read segment from the queue (more segments still exist for message).
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.

4.21 iqr_reset_stat_q

This service is used to reset statistical counters and timers for a message queue.

FORMAT

iqr_reset_stat_q (hub, queue_name)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of read, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read
Mechanism:	By reference

This is the name of the message queue to reset statistics for.

DESCRIPTION

This service will reset statistics for a message queue. All timers for reads, writes, acknowlegment, and cumulative timers will be reset to a time of zero. In addition, the number of reads and writes for the message queue will be reset to zero.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successfully read segment from the queue (more segments still exist for message).
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.

4.22 iqr_rtr_write_q

Write a message to a message queue using a buffer formatted with header information.

FORMAT

iqr_rtr_write_q (hub, queue_name, buffer, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of write, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to write to.

buffer

Type:	Descriptor
Access:	Read
Mechanism:	By reference

This is a descriptor for the buffer where the information is to be placed on the queue. This buffer contains both a filled header (of type HDRDEF) followed by its message. *The descriptor length must include the length of the header* (HDR$K_SIZ), but the buffer location should point to the actual message. The header portion must immediately precede the message buffer address passed.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This routine will attempt to write a new message to a message queue using the provided message header information passed in the buffer.

The caller must have already connected either for read or write (both allow write access). The message in the user's buffer is then copied onto the queue to await for reading. If there were no messages currently waiting in the queue, then the routine will attempt to notify any processes that are currently waiting for a new message to arrive.

Note that there are two limits to the number of messages that can be written to the queue. At any one time, there is a maximum size to both the messages allowed and the size of the queue itself. If your message exceeds either of these limits, then an error is returned. Also, the message queue was set up with a maximum number of messages allowed. Exceeding this value will also return an error condition.

This particular routine is similar to **iqr_write_q**, but differs in that the caller must pass both the header information and message to the service. Use this service if you have a header for a message that you want to preserve along with the message. Pay special attention to the format of the buffer (see above)! This service will preserve the header onto the queue, updating only necessary information. This service is usually called by the router when moving a message from another router.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successful completion.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.

QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_INVALUSERBUF	The buffer passed by the user is invalid.
QUE_NOTCONWRITE	The caller has not yet connected for write (or read) to this queue.
QUE_MQDFULL	The message queue is full (the number of messages in the queue exceeds that set by iqr_add_message_q).

4.23 iqr_thread_msgblks

Thread allocated and filled message blocks into a message queue.

FORMAT

iqr_thread_msgblks (hub, queue_name, hmb_blck, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of threading, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to use.

hmb_blck

Type:	Longword
Access:	Read
Mechanism:	By value

This is the value of the block number of the header message block for the message that is to be threaded.

queue_index

Type:	Longword
Access:	Read
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This service entry is called by the router or another process that needs to thread a message that has been filled into the message queue's message thread. This service requires that the user has the SYSNAM privilege. This test is to ensure that the caller is a knowledgeable user. The privilege is not actually needed by the service.

The blocks of the message have been assumed to have been allocated using the **iqr_allocate_msgblks** and populated with the **iqr_fill_msgblks**.

Note: Since the test for full message queues is done at the time of allocation, it is possible to end up with a message queue that has more messages than allowed. This should not; however, affect the overall operation of the queue service.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successful return.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_NOTCONWRITE	The caller has not yet connected for write (or read) to this queue.
QUE_MQDFULL	The message queue is full (the number of messages in the queue exceeds that set by iqr_add_message_q).

4.24 iqr_write_q

Write a message to a message queue.

FORMAT

iqr_write_q (hub, queue_name, buffer, queue_index)

RETURNS

VMS usage:	Condition code
Type:	Longword
Mechanism:	By value

Result of write, including possible VMS and RMS status codes.

ARGUMENTS

hub

Type:	Record structure hbkdef
Access:	Read
Mechanism:	By reference

This is the buffer that was returned by the **iqr_attach_h** service. This is defined by the structure HBKDEF in the file HBKDEF.H

queue_name

Type:	Descriptor
Access:	Read only
Mechanism:	By reference

This is the name of the message queue to write to.

buffer

Type:	Descriptor
Access:	Read
Mechanism:	By reference

This is a descriptor for the buffer where the information is to be placed on the queue.

queue_index

Type:	Longword
Access:	Write
Mechanism:	By value

This is the Connected Message Queue (CMQ) value returned by the **iqr_connect_read** or **iqr_connect_write** service routines for the message queue.

DESCRIPTION

This routine will attempt to write a new message to a message queue.

The caller must have already connected either for read or write (both allow write access). The message in the user's buffer is then copied onto the queue to await for reading. If there were no messages currently waiting in the queue, then the routine will attempt to notify any processes that are currently waiting for a new message to arrive.

Note that there are two limits to the number of messages that can be written to the queue. At any one time there is a maximum size to both the messages allowed and the size of the queue itself. If your message would exceed either of these limits, then an error is returned. Also, the message queue was set up with a maximum number of messages allowed. Exceeding this value will also return an error condition.

If the message queue is full, then it is checked for any stale or volatile messages that can be deleted before writing to the message queue. Deleted messages cannot be recovered.

CONDITION VALUES RETURNED

VMS Condition Codes

QUE_SUCCESS	Successful completion
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_INVALIDPEX	User passed hub was invalid. Missing or not writeable.
QUE_INTERNALFAULT	Contact developers.
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_INVALUSERBUF	The buffer passed by the user is invalid

QUE_NOTCONWRITE	The caller has not yet connected for write (or read) to this queue.
QUE_MQDFULL	The message queue is full (the number of messages in the queue exceeds that set by iqr_add_message_q).

5. Return Status Codes

This is a summary of status condition codes returned from IQR System Service routines.

5.1 Successful Status Codes

VMS Error Code	***Description***
QUE_ADDED	A new message queue was successfully added. Only returned from **iqr_add_message_q**.
QUE_LASTSEG	Successful read of a segment from the queue (no more segments exist, must ACK message before reading again). Only returned from **iqr_read_segment**.
QUE_SUCCESS	Normal, successful return.

5.2 Failure Status Codes

VMS Error Code	***Description***
QUE_ALLOCLOCK	Unable to capture hub allocation lock. Examine extended status in hbk and contact developers.
QUE_BADCCTMQD	The user specified number of message queues exceeds a reasonability test defined by the software. Contact developers if this is too low for your environment. The maximum number was specified when the software was packaged.
QUE_BADHNAME	User passed hub name was invalid. Name was too short, too long, or was not accessible for read.
QUE_BADPRCINF	The queue service was unable to get information about the current process. Examine the condition code in the extended status field of the hbk. Contact the developers.
QUE_CONTAINERFULL	The disk container is at its maximum size. Either remove waiting messages in a message queue, remove a message queue, or increase the size of the container file.
QUE_DEFHNAME	Unable to translate the default hub name. IQRHUB logical not defined by the system manager.
QUE_INTERNALFAULT	Contact developers.
QUE_INVALIDPEX	User passed hub value was invalid. Missing or not writeable. Usually caused by not having the hub initialized through the **iqr_attach_h** routine.

QUE_INVALQIDX	The queue_index is not valid. Usually caused by not successfully connecting to a message queue via **iqr_connect_read** or **iqr_connect_write** or trying to access a queue after disconnecting.
QUE_INVALQNAME	User passed message queue name was not valid. Not readable, or long length. Can also indicate the message queue name does not match that given by the queue_index.
QUE_INVALUSERBUF	The buffer passed by the user is invalid. Either too small or not a valid memory index passed.
QUE_INVARG	User passed argument invalid, or not readable.
QUE_MAXMSGQUEUES	User has exceeded the maximum number of message queues that may be attached. The number specified on the **iqr_attach_h** or **mqd_count** must be increased.
QUE_MQDFULL	The message queue is full (the number of messages in the queue exceeds that set by **iqr_add_message_q**).
QUE_NOCACHE	No cache space available for the message queue.
QUE_NOMESS	The message queue is currently empty. No messages exist to read.
QUE_NORNAMESS	No message was read off of the queue -- the RNA has not been set. You need to first read a message before you can acknowledge it.
QUE_NOTCONREAD	The caller has not yet connected for read to this queue.
QUE_NOTCONWRITE	The caller has not yet connected for write to this queue.
QUE_NOTFOUND	Requested message queue was not found within the current hub.
QUE_PRCLCKNM	The queue service was unable to create a process lock. Examine the condition code in the extended status field of the hbk.
QUE_PREATT	Informational status indicating that the message hub has already been attached.
QUE_TOOMANYRDR	Maximum number of readers already connected to the message queue.
QUE_USRBUFSML	The size of the passed buffer is too small.

6. Using the System Services

6.1 Code Generation

While the bulk of the IQR software was written in DEC C, header files are provided for both C and FORTRAN. Specific information relative to each programming environment is provided in following sections.

Program code written to use the IQR System Service usually follows some basic patterns in order to read/write to message queues. They are:

- First, attach to a specific hub by calling **iqr_attach_h**. The returned *hub* value must be saved for future service calls that reference this particular messaging hub.
- Insure that the message queue you are going to use exists. If it does not, you can create it with the IQU utility, or call the **iqr_add_message_q** service. If the message queue does not exist, then any attempts to connect to the message queue will fail.
- Connect for read or write to a particular message queue, depending on what you intend to do. The number of readers on a queue is limited, so unless you intend to read from a queue, select to connect as a writer. Readers can both read and write messages. Call **iqr_connect_read** or **iqr_connect_write** according to your intentions. Be sure to save the returned *queue_index* value for future service calls that refer to these message queues.
- Actually performs the read/write. Reading can be done with a number of services such as:
    - ◊ **iqr_read_q** - read a message from the queue (normal read).
    - ◊ **iqr_read_qw** - read a message from the queue. If none exists, wait for one to arrive.
    - ◊ **iqr_read_segment** - read a part of a message from the queue. Successive calls to this service are needed to read in the entire message.
    - ◊ **iqr_read_hmb** - call this service to just check the information about the next message on the queue. The message remains at the head of the queue.

  After reading from the queue, you will need to acknowledge the message. This lets the queue know that you are done with the message and may remove it from the message queue. You may not have to do this if the message queue was set up as *auto acknowledging* (see **iqr_modify_q** or **iqr_add_message_q**). Use the routine **iqr_ack_read** to acknowledge the message.

  Writing a message can be done with these services:

    - ◊ **iqr_write_q** - the normal write routine
    - ◊ **iqr_rtr_write_q** - this will write a message to a queue, preserving an already built header for the message.
    - ◊ **iqr_allocate_msgblks, iqr_fill_msgblks,** and **iqr_thread_msgblks** - these services combined allow the caller to fill a message on the queue using separate segments.

- When you have finished working with a specific message queue, you should disconnect from it (especially if you are a reader; other readers may need to connect). Call **iqr_disconnect_q** to disconnect yourself from the message queue. Further access to the message queue will require that you reconnect again. If you are completely done with a message queue, or as a rundown service, you may wish to execute the service **iqr_disconnect_h** which disconnects the program from the hub and shutdown all opened message queues.

6.2 Using IQR with C

The following suggestions are for people who wish to program in C:

- In source code that uses IQR routines or definitions, you need to include the following header file:

```
#include IQR
```

- When compiling programs that use the IQR system service, you need to make reference to the header library. Do this by:

```
CC <source> <options> + IQR$LIB:HUB.TLB/LIBRARY
```

- To link programs to the IQR system service, do the following:

```
$ LINK <source> SYS$INPUT/OPTIONS
   <options>
   IQR$LIB:HUB/LIBRARY
   IQR$PROD:IQRSS/SHARE
```

6.3 Using IQR with FORTRAN

The following suggestions are for people who wish to program in FORTRAN:

- In source code that uses IQR routines or definitions, you need to include the following header file:

```
INCLUDE 'IQR$LIB:HUB_FOR(IQR)'
```

- When compiling programs that use the IQR system service, you need to cancel warnings about structure alignment. Do this by:

```
FORTRAN /NOWARN=ALIGN <options> <source>
```

- To link programs to the IQR system service, do the following:

```
$ LINK <source> SYS$INPUT/OPTIONS
   <options>
   IQR$LIB:HUB/LIBRARY
   IQR$PROD:IQRSS/SHARE
```

7. Compatibility

The IQR System Services includes a patch library that will enable users to link older MAQ or MQD software to the current version of the IQR System Service. The IQR Service and Router were designed to be compatible with these previous queuing software programs. However, complete compatibility is not entirely possible, and some general notes for both MAQ and MQD versions of software are given below:

- The IQR System Service requires that the user pass an *index value* in the service calls. In order to provide compatibility, the patch library will check for a valid index argument, and if not found, will attempt to search the callers connected message queues for a valid index. If the index is still not found, then an attempt will be made to connect to the message queue to get an index. If the user is a reader of a message queue, he *must* connect before attempting to read from the queue -- the patch library will not perform a connect_read.
- The IQR System Service no longer provides the user with a valid RNA value. This number is now kept internally by the IQR software. To provide compatibility, the patch routines will return a value of one (1) for all routines that return a valid RNA value.
- MAQ/MQD System calls that relied upon the gathering of information from the message queue will not work under the new IQR System Service -- mainly due to the incompatibility of how the message queues are actually stored on each software platform and the format of data structures. If you desire information about a message queue, use the provided IQR System Service utilities or calls.
- MAQ/MQD System calls that actually perform message queue functions (i.e. read, write, delete, attach) are all supported through the use of a patch library that is included along with the IQR System Service. Original source code will need to be re-linked (and possibly recompiled) with the new patch library in order to work with the IQR System Service.

Specific information regarding the two versions of queuing software is described below.

7.1 MAQ System Service Patch Library

The patch library for the MAQ System Service supports MAQ version 5.3. For more specific information of function calls, see the MAQ manual. The following function calls are currently supported:

MAQ Service Calls	***Notes***
ack_read	Acknowledge last read message.
ack_sec_read	Outdated service. Calling this routine will actually perform *ack_read*.
add_message_id	Adds a new message queue to a hub with default parameters except for the *max_count* parameter.
attach_q	Will attach to a messaging hub. For a default hub name, you must have IQRHUB defined in either the group or system tables.
attach_qe	Identical to *attach_q*.
backup_rna	Backup the RNA of the current read message.
backup_srna	Outdated service. Calling this routine will actually perform *backup_rna*.
change_message_id	Performs a limited form of the iqr_modify_q service. This will only allow the caller to modify the maximum number of messages in a queue at one time.
con_secread	Outdated service. Calling this routine will actually perform *connect_read*.
connect_read	Connects the caller as either a primary or secondary reader to a message queue.
delete_message_id	Delete a message queue from a hub.
detach_q	Detaches the current process from all connected message queues (rundown handler).
disconnect_read	This will disconnect a reader (or writer) from a message queue.
get_mid_index	Actually performs an iqr_connect_write. This is used to get an index value for writers to a message queue.
read_q	Reads a message from a message queue.
read_qrec	Read a message from the queue (record format).
read_sq	Outdated service. Actually performs the *read_q* service.

write_q	Writes a message to a message queue.
write_qrec	Writes a message to a message queue (record format).

The following MAQ routines are not supported and will return a QUE_NOTSUPP warning error.

Unsupported MAQ Service Calls	***Notes***
display_message_id	Invalid data under IQR.
display_queue_head	Invalid data under IQR.
display_region	Invalid data under IQR.
find_q_processes	Not supported in IQR.
opr_fao_msg	Not supported in IQR.
read_msg	Not yet supported.
rtr_write_q	Not yet supported.
shutdown_q	Not yet supported.

To compile MAQ Service programs, the following must be done:

- Compile all source code. In particular, on the Alpha platform. It is imperative that some of the data structures be aligned properly. This usually will either require a special command line switch or a command statement in the source code. The following data structures ***must*** be compiled so as to be ***byte*** aligned, otherwise strange data and errors may appear (particularly when dealing with messages going to/from the router):

  ◊ HDRDEF

  If you need to know how to compile a module so as to be aligned, see your compiler's documentation. For FORTRAN code, use the following technique for the included file:

```
cdec$options/align=(record=packed)       !Turn on byte alignment
      INCLUDE 'QUEUE.TLB(HDRDEF)'        !Aligns this module
   cdec$end options                      !Restore to normal alignment
```

  In addition, add the `/NOWARN=ALIGN` switch to the FORTRAN compiler command line to disable reports about misalignment.

- Include the IQR Patch MAQ Library (iqr_patch_maq), IQR System Service (iqrss), and service messages library (hub) in the link statement of your program. This should be like the following:

```
$ LINK <source> SYS$INPUT/OPTIONS
      <options>
      IQR$LIB:PATCH_MAQ/LIBRARY
      IQR$PROD:IQRSS/SHARE
```

7.2 MQD System Service Patch Library

The patch library for the MQD System Service supports MQD version 4.0. For more specific information of function calls, see the MQD manual. The following function calls are currently supported:

MQD Service Calls	***Notes***
mqd$ack_read	Acknowledge last read message.
mqd$attach_q	Will attach to a messaging hub. For a default hub name, you must have IQRHUB defined in either the group or system tables.
mqd$attach_qe	Identical to *attach_q*.
mqd$backup_rna	Backup the RNA of the current read message.
mqd$connect_read	Connects the caller as either a primary or secondary reader to a message queue.
mqd$detach_q	Detaches the current process from all connected message queues (rundown handler).
mqd$get_mid_index	Actually performs an iqr_connect_write. This is used to get an index value for writers to a message queue.
mqd$ack_read	Acknowledge last read message.
mqd$add_message_id	Adds a new message queue to a hub.
mqd$attach_q	Will attach to a messaging hub. For a default hub name, you must have IQRHUB defined in either the group or system tables.
mqd$backup_rna	Backup the RNA of the current read message.
mqd$change_message_id	Modifies the number of messages allowed on a message queue.
mqd$connect_read	Connects the caller as either a primary or secondary reader to a message queue.
mqd$connect_write	Connects the caller as a writer to a message queue.
mqd$delete_message_id	Deletes a message queue from a hub.
mqd$detach_q	Detaches the current process from all connected message queues (rundown handler).
mqd$disconnect_id	Disconnect from a message queue.
mqd$read_q	Reads a message from a message queue.
mqd$read_qn	Reads a message from a message queue and if no message is present, will trigger a user written AST when a message arrives.
mqd$read_qw	Reads a message from a message queue (wait for message to arrive).
mqd$write_q	Writes a message to a message queue.
mqd$read_q	Reads a message from a message queue.
mqd$read_qrec	Read a message from a message queue (record format).
mqd$write_q	Writes a message to a message queue.
mqd$write_qrec	Write a message to a message queue (record format).

7.3 BEA Message Q

BEA Message Q, formerly DEC Message Q (DMQ) provides much of the same functionality as IQR Services do. Currently there is no compatibility library for DMQ but one could easily be created. Differences that would need to be addressed are:

Message Bus- This would map to a Hub in the IQR environment
Cross Bus Connections- A router process would provide this cross over
Different message classifications- Class and type message type are provided by DMQ. The DMQ allows for selective delivery of message by class. This would need to be provided by a second message queue in the IQR environment.
Numeric Message Queues- DMQ uses numeric message queues while the IQR Services uses symbolic message queues and numeric once a queue is connected. One could maintain a simple cross reference for these as the DMQ$INIT.TXT file does or one could change the prototype for the DMQ call and the definition of the DMQ text include file used by user code and actually use a symbolic queue name like IQR does.

7.4 Microsoft Message Queue

Microsoft Message Queue also provides a similar service to IQR. A router running in the Microsoft Windows environment could easily bridge the IQR and MSMQ queues. Currently IQR only runs on OpenVMS Itanium and OpenVMS Alpha (it has never been compiled on OpenVMS VAX, but this should not be a difficult task).

The following MQD routines are not supported and will return a QUE_NOTSUPP warning error.

Unsupported MQD Service Calls	***Notes***
mqd$attach_d	Invalid data under IQR.
mqd$cplx_time	Not supported in IQR.
mqd$display_message_id	Invalid data under IQR.
mqd$display_queue_head	Invalid data under IQR.
mqd$display_region	Invalid data under IQR.
mqd$rtr_write_q	Not yet supported.
mqd$set_ctime	Not supported in IQR.

To compile MQD Service programs, the following must be done:

- Compile all source code. In particular, on the Alpha platform, it is imperative that some of the data structures be aligned properly. This usually will either require a special command line switch or a command statement in the source code. The following data structures ***must*** be compiled so as to be ***byte*** aligned, otherwise strange data and errors may appear:

  ◊ HDRDEF

  If you need to know how to compile a module so as to be aligned, see your compiler's documentation. For FORTRAN code, use the following technique for the included file:

```
cdec$   options/align=(record=packed)         !Turn on byte alignment
        INCLUDE 'QUEUE.TLB(HDRDEF)'           !Aligns this module
cdec$   end options                           !Restore to normal
```

  In addition, add the `/NOWARN=ALIGN` switch to the FORTRAN compiler command line to disable reports about misalignment.

- Include the IQR Patch MQD Library (iqr_patch_mqd), IQR System Service (IQRSS), and service messages library (hub) in the link statement of your program. This should be similar to the following:

```
$ LINK <source> SYS$INPUT/OPTIONS
   <options>
   IQR$LIB:IQR_PATCH_MQD/LIBRARY
   IQR$PROD:IQRSS/SHARE
```

8. IQR Router

8.1 Introduction

The IQR Routers provides the ability to route message queues to other nodes. Currently, DECnet and TCP/IP transports are supported. The remote nodes may be any DECnet or TCP/IP compatible node that supports the IQR Router protocol. This protocol and example test programs are available from IPACT to any end user who desires to write their own router to communicate with the IQR Router.

The TCP/IP IQR Router can be used in conjunction with the MAQ product as well as with the IQR product. MAQ is a package available to the public through the DECUS organization. It provides a queuing service to OpenVMS VAX environments. The TCP/IP IQR Router can be installed on a system having the MAQ product, and will route the MAQ messages to other nodes supporting the IQR Router Protocol.

Applications and their message queues should be partitioned such that a particular router may be taken down while still allowing other applications to function.

8.2 TCP/IP IQR Router

The TCP/IP IQR Router is a threads based application that facilitates the transmission of queue messages from one hub/node to another over a standard TCP/IP connection. The router uses standard Posix Compliant thread calls and standard TCP/IP socket services. This router variation has been tested on OpenVMS VAX and OpenVMS AXP. OpenVMS must be at least version 6.2 to support the threads environment. Two TCP/IP stacks have been tested with this version of TCP/IP Router. DEC (Compaq) TCP/IP Services version 4.2 or greater and Process Software Corporation's TCPWARE version 5.2 or greater have been used successfully with this product. Other stacks should function as well, if they provide a standard socket library to the OpenVMS environment.

The TCP/IP IQR Router is provided as a standard part of the IQR product. It can also be used in the MAQ environment by obtaining a recent version of the MAQ kit from IPACT. You will then be able to route messages between IQR and MAQ over TCP/IP in addition to DECnet.

8.3 DECnet Router Routing Database

The IQR Router uses a routing database that specifies which message queues are to be received by a particular router and which are to be transmitted by a router. Each IQR Router has its own unique routing database. Multiple routers may be present on a single node such that applications can be partitioned. Each router has the ability to connect to one or more hubs (beta release only supports a single hub connection). A router is known to other routers by its node name and its object name. Object names should reflect the functionality of the router. DECnet requires that object names be unique.

To generate a routing database for a router the following language is provided. The language is then compiled and built into a routing database image by the RTRDBS utility. The RTRDBS utility also creates a startup command procedure for the router that can be used by the system manager to start the particular IQR Router. This command procedure specifies adequate resources needed by the router based on the information contained within the routing database.

The following four statements that are used to define the routing database are as follows:

- **ROUTER** - Specifies information for the IQR Router process
- **NODE** - Specifies connection to remote nodes
- **ROUTE_QUEUE_OUT** - Specify message queues to be routed off this node by this router
- **RECEIVE_QUEUE** - Specify message queues to be received by this router

Each of the statements are shown below with their syntax. All parameters are separated by commas and are free of format. All lines beginning with an exclamation point and the remainder of any line after an exclamation point is considered to be a comment. Parameters enclosed by square brackets "[]" are optional and a default value will be supplied by the RTRDBS utility. All statements must end with a semicolon. The language is translated to upper case prior to parsing. Therefore, all message queues, process names, object names, and hub names are all converted to upper case. The user may not use any of the reserved words shown in capital letters.

ROUTER
OUTBUF_SIZE=blkcnt,
OUTBUF_COUNT=bufcnt,
PROCESS_NAME=pname,
DENCET_OBJECT_NAME=objname,
[**QUEUE_NOACK_TIMER**=tvalue];

blkcnt = Size of the output buffers. This number is specified in 512 byte blocks (e.g. a value of 2 specifies a 1024 byte buffer)

bufcnt = Number of output buffers. This effects the number of messages that may be transmitted by the router at a single time. It also effects the buffer I/O quota required by the router.

pname = Process name that the router should define itself. The router will change its process name to this name when it begins running. This is done to ensure that there can never be two routers running against the same routing database. Standard VMS process names are valid.

objname = This is the DECnet object that will be mapped within the router database that is used to identify itself to DECNET. This name must be unique. The system manager can use the following VMS command to determine if the object name is unique: "NCP SHOW KNOWN OBJECTS".

tvalue = This is a timer specified in seconds that indicates how long a message written to a remote node will be considered, not acknowledged. This value is defaulted to fifteen seconds.

NODE
NODE_NAME=logical node name,
NODE_LIST=(node1[,node2,node3,node4]),
OBJECT=object name,
MAX_RECV_MSG_SIZE=blkcnt,
[**FLAGS**=(flag1,flag2)],
[**RETRY_TIMER**=rtime];

logical node name = This is a logical name for a node or nodes where the local router may send outbound message queues. Normally, this is specified as a service name on the remote node. Typical names might be: "lineups, production, development". This name must be sixteen characters or less.

(node1,..,node4) = This is a list of DECnet node names of where a remote router might exist. This list may be from one to four in number. The router will attempt to connect to the object name indicated at each of the nodes with a one minute interval between attempts. If the DECnet object is not available on any of the nodes, then the router will delay for "RETRY_TIMER" minutes before trying the list again.

object name = This is the DECnet object on each of the remote nodes where the router should attempt to connect. The router uses the same object name for each of the nodes in the NODE_LIST.

blkcnt = This is the largest message that may be received from the remote node in 512 byte blocks.

(flag1,..,flagn) = These are character flags that are used by the router for particular functioning to a remote router. Currently, the following flags are defined:

- **RSX** - If this flag is set, the router will use the non-multipacked router protocol used by the MAQ router on the RSX platform or the MAQ router for VMS releases less than 5.3. The MAQ router for RSX is available from IPACT or DECUS.
- **MQD** - If this flag is set, the router will use the multipacked router protocol used by the MQD router. The MQD router is a proprietary router developed by IPACT for Inland Steel.

rtime = Time in minutes between attempts of the route list. If none of the nodes in the route list are found to be reachable or are unable to connect to the remote router on any of the nodes in the route list, then the router will wait this amount of time before trying the list again. The default value is fifteen minutes.

ROUTE_QUEUE_OUT
QUEUE_NAME=message queue,
FROM_HUB=hub,
TO_NODE=logical node name,
[**FULL_TIMER**=ftime,]
[AS_QUEUE=alternate message queue name];

message queue = This is the name of a message queue that the router will connect as a reader to and attempt to route to a remote router.

hub = This is the hub where the message queue resides.

logical node name = This specifies which node the messages contained in the message queue are to be routed. There must be a node statement with the "NODE_NAME" specified to this.

ftime = If the router receives a response from a remote router that its hub is full, then the router will wait this amount of time before trying to send this message to the remote router again.

alternate message queue name = This allows the ability to change the name of the message queue when it is routed off node.

RECEIVE_QUEUE
QUEUE_NAME=message queue,
TO_HUB=hub

message queue = This is a message queue that this router should expect to receive from any of the nodes that connects to it.

hub = This is the hub where the router should place the message queue when it is received.

A sample routing database follows:

```
!
! -- A sample router database file for use with the TEST_IQR hub --
!
!
! This section defines global parameters for the router that uses this
! router database.
!
ROUTER
        OUTBUF_COUNT=4,                         ! Number of output buffers
        OUTBUF_SIZE=4,                          ! Size of output buffers in 512Kb blocks
        PROCESS_NAME=TEST_RTR,                  ! Name of the process when running
        DECNET_OBJECT_NAME=TEST_RTR,            ! DECNET object name of this router
        QUEUE_NOACK_TIMER=60;                   ! Timer if a message is sent to a remote
                                                ! node and no acknowledge is received (in
                                                ! seconds)
!
! This section defines the NODE names that the router can connect to.  Each node
! name actually can have up to four named nodes in it.
!
NODE
    NODE_NAME= MV3,          ! Define remote logical node service as this name
    NODE_LIST=(IPCMV3),      ! List of nodes in this group that we try to connect
    OBJECT=IPCMV3_RTR,       ! Remote router object name is this
    MAX_RECV_MSG_SIZE=10,    ! Largest message to be received
                             !  from this node is in 512 Kb blocks
    FLAGS=(MQD),             ! Some flags
    RETRY_TIMER=15;          ! And if we can't connect any of the nodes in
                             ! route list, how long to wait before
                             ! trying the route list again (in minutes)
!
! This section defines outbound routed message queues.  You must have a separate
! entry for every message queue to be routed out.
!
ROUTE_QUEUE_OUT
     QUEUE_NAME=MSG_OUT,         ! Msg queue from this hub to be routed outbound
     AS_QUEUE=MSG_OUT,           ! Routed queue name
     FROM_HUB=TEST_IQR,          ! and from this hub
     TO_NODE=MV3,                ! to this target node
     FULL_TIMER=120;             ! if full, how long before trying again in seconds
!
! This section defines a message queue that is capable of receiving routed
! messages from another router.  You must have a seperate section for each
! message queue to be received.
!
RECEIVE_QUEUE
    QUEUE_NAME=MSG_IN,           ! Receive into this message queue from anyone
    TO_HUB=TEST_IQR;             ! Which is located in this hub
```

8.4 DECnet Routing Utilities

To help diagnose the actions of the router the following utilities are supplied:

- **DMPRTR** - This utility will display the connection status of all the remote routers. It will also display statistics for each of the paths.
- **LSTRTR** - This utility will display the message queues transmitted or received by the router.

These utilities are fully described in the Utilities Chapter.

8.5 TCP/IP Router Routing Database

The function of the TCP/IP Router's Routing Database is similar in nature to the one used by the DECnet Router. The main difference between the two being, the method used to generate the actual database. The database, or initialization file, used by the TCP/IP IQR Router is a simple text file which is created using your favorite text editor. Its structure is similar to the structure used in many Windows based applications having initialization files.

The database is created by opening a standard text file using a text editor. The name of the file is usually chosen to be representative of the environment to be serviced by the specific router. The length of the file name is limited only by the operating system on which the router will be running.

The database file is partitioned into three major sections. Each major section name is delimited by a pair of open an closed square brackets "[ ]". Section parameter values are specified following each section identifier by using a set of predefined and unique keywords. Each section and each keyword for these sections is discussed in the following paragraphs. Each keyword is assigned a value by forming a definition such as HUB = ABCDEF_GH or PORT = 12345.

The following three sections are used in combination to define the TCP/IP routing database:

- **GLOBAL** - Specifies information for the IQR Router process
- **INCOMING** - Specifies connection to remote nodes
- **OUTGOING** - Specify message queues to be routed off this node by this router

References to "Nodes" in the context of the TCP/IP IQR Router refer to a name entry in the hosts file of the local node. These names are subsequently translated into complete TCP/IP addresses by use of standard socket service calls. Relationships may be established, by the System or Network administrator, which result in specific paths being used for connections to remote routers. Please consult your TCP/IP stack provider's management guide or contact your administrator for assistance in these areas.

8.5.1 TCP/IP Router Database [GLOBAL] Section

The first section to be defined in the Routing Database of a TCP/IP IQR Router is the [Global] section, not to be confused with a global section of memory. This section of the Router Database, or initialization file, describes the hub to which this instance of router will attach and then service. The Global section has only 3 parameters. All three of these parameters are optional. If they are not explicitly defined, a default is used for the parameter.

Hub -The first of the optional parameters for the [GLOBAL] section is the HUB parameter. This parameter identifies the hub to which this instance of router is to attach. If the hub parameter is not defined, the logical name IQRHUB is translated to obtain the default hub name on this node. The hub name is limited to eight characters in length.

Port -The next [GLOBAL] parameter is the TCP/IP port on which to listen for incoming connections from other routers or utility applications such as TCPIQRSTAT. If a port is not specified in the global section of the initialization file, a default of port 3000 is used. If a port other than 3000 is used, the port number must also be specified in utility operations as well. TCPIQRSTAT will use port 3000 by default. When choosing a port number it is best to check with the system or network administration for your network. You must insure that the chosen port number is not used by any other applications on your network. Unexpected behavior will result if other applications are using the same port number.

BufferSize -The last parameter which can be specified for the Global section is BufferSize. This parameter is only used in environments which are using the MAQ product mentioned previously in this document. The MAQ product is only able to deliver a complete queue message to the router when read, unlike the IQR product which feeds the router smaller portions of a queue message for processing. Given this mode of operation by the MAQ, it is necessary that the TCP/IP IQR Router have sufficient buffer space to accommodate the largest message being queued within the MA Queue. If a value is not provided in the initialization file, a default value of 8192 bytes is used by default. The minimum size that may be specified is currently 1024 bytes. The largest buffer that may be specified is currently 32767 bytes. If the router is being used in an IQR environment, the buffersize parameter is ignored. Calculate the correct buffersize by determining the largest message contained in the hub and adding 106 bytes to that value. This is the correct size for parameter BufferSize.

8.5.2 TCP/IP Router Database [INCOMING] Section

The next section of the TCP/IP Router Database is the Incoming section. It defines a list of queues to which incoming messages are expected to be routed to, from other routers. A queue specified in this section is attached to by the router for write access. Only a single parameter is supported in the Incoming section of the initialization file.

QueueName – This parameter defines the name of a message queue, in the attached hub, to which this instance of the router will write messages. Any number of QueueName entries may follow the Incoming section heading. The number of entries is limited only by the number of queues defined in the attached hub. The remote router sending messages to this queue will have a corresponding entry in the Outgoing section of its initialization file.

8.5.3 TCP/IP Router Database [OUTGOING] Section

The last section of the TCP/IP Router Database is the Outgoing section. This section defines those queues, whose messages are destined for other cooperating TCP/IP Routers. Entries in this section describe local message queues and the destination routers for the messages contained within them. There are six parameters supported in the Outgoing section of the initialization file. The first two parameters are required for each outgoing queue. The next four parameters are optional.

QueueName – This is the first parameter to be defined for an entry in the Outgoing section of the initialization file. It specifies the name of a queue whose messages are destined for another hub. The TCP/IP Router is notified when a message is deposited in the specified local message queue. The router is then responsible for delivering the message to the specified cooperating router for entry in that router's target hub.

RemoteNode – The next parameter used in defining an outgoing queue is the RemoteNode parameter. It is a required parameter for an outgoing queue. The parameter can be used in two distinct ways. If the parameter is used horizontally, such as RemoteNode = NodeABCD, NodeJKLM, the nodes specified on the line are treated as primary and secondary. If a send to the first node is unsuccessful, a connection is made to the secondary node and an attempt is made to send the queued message to that node. Attempts are then made at defined intervals to these nodes, in a round robin fashion, until a successful send occurs. That node then becomes the current primary node for that message queue.

If multiple RemoteNode entries are present for an outgoing message queue, each node in the list is sent a copy of the message being routed, if the number of nodes connected is greater than or equal to the number specified by parameter **Mincon**. If that number of connections have not been established for the current outgoing queue, no messages are sent. When the specified number of connections are finally made, the messages are then transmitted by the router to the list of nodes.

You may also use a combination of the two RemoteNode specifications. You may choose to have multiple RemoteNode lines specified for an outgoing queue in addition to supplying a many as two nodes per RemoteNode parameter line.

Bear in mind that a TCP/IP socket is consumed for each active connection to a remote router from the local router. If you have you have defined eight incoming and eight outgoing queues you have just consumed a minimum of 16 TCP/IP sockets for this configuration. If you have specified multiple RemoteNode parameter lines for outgoing queues you have increased the original number of sockets consumed by the number of additional RemoteNode lines present. An additional socket is used by the main router thread to listen for incoming connect request from remote routers. One final socket us used by the TCPIQRSTAT utility in order to obtain routing information from a TCP/IP router. Attention must be paid to the number of sockets being consumed as the limit on socket usage for the router at present is 64. Exceeding this number will result in unpredictable behavior of the router and message delivery.

RemotePort – The RemotePort parameter designates the port to which a connection should be made at the remote router. If not specified, the default port of 3000 is used. This is an optional parameter.

RemoteQName –The RemoteQName parameter is used when the target queue at the remote end differs from the queue name locally. Under normal circumstances the specified QueueName is also used as the name of the target queue at the remote destination. In fact that is the default case when the RemoteQName parameter is omitted. This is an optional parameter.

RetryTimer – The RetryTimer parameter is used to set the retry interval for reconnection attempts between the local router and the remote router for this queue. The parameter is specified as an integer number of minutes between retry attempts to a disconnected router. This parameter is optional. If unspecified, a default of 1 minute is used.

MinCon – The MinCon parameter specifies the minimum number of remote connections required before forwarding of messages from this queue will begin. This parameter is especially important when using multiple RemoteNode parameter lines for a single outgoing queue. This parameter is optional. If unspecified, a default of 1 is used for the parameter. An example of its use would be specifying three RemoteNode parameter lines each with a single node identified. If a MinCon value of 2 is specified, the router must have established a connection with at least 2 of the three routers specified in the RemoteNode parameter lines before any message will be forwarded from the local message queue.

Example of TCP/IP IQR Router Database initialization file:

```
!*************************************************************************
! This version of the TCPIQR initialization file was created as an example *
! of how you might structure your initialization file.                     *
!*************************************************************************
! TCP/IP IQR Router definition file
!
! This file is read during startup of the TCPIQR process to configure
! global buffers for local and remote queue access. The file has a
! structure similar to other Windows based application initialization files
! as shown below.
!
! Specify sections with brackets used to designate the beginning of a
! section, such as: [SECTION_NAME].  Valid sections are GLOBAL,
! INCOMING, and OUTGOING
!
! Specify section parameters as PARAMETER = VALUE
! You may have spaces/tabs around both PARAMETER and VALUE.  Leading
! and trailing spaces are stripped by TCPIQR. Valid paramters for the
! various sections are shown below.
!
!
! GLOBAL parameters.  All are optional.
!
! Parameters:
! HUB   name of hub to attach to. If not supplied use logical name IQRHUB
!       to derive the hub name.
! PORT  is the port number on which the router listens for incoming
!       connect requests. If not supplied, the default port of 3000 is used.
! BUFFERSIZE
!       Only used on MAQ based systems to specify largest expected message
!       to be read from a local queue. It is optional. It is ignored on
!       IQR based systems. 8192 is the default buffersize when not specified.
!
[GLOBAL]
HUB=TEST_HUB
!
! Incoming section.  Specifications in this section pertain to local queues
! which are to be written to by other routers. These routers may be located
! on remote nodes or the same node.
!
!
! INCOMING parameters. Only a single parameter is currently supported.
!
! Parameters:
! QUEUENAME Is the name of the local que to be written to by the router
!           upon receipt of a message designated for that queue.
!
[INCOMING]
QueueName=LOCAL_QUE1             ! Incoming queue name
QueueName=LOCAL_QUE2             ! Incoming queue name
QueueName=LOCAL_QUE3             ! Incoming queue name
!
! OUTGOING Parameters.  Specify a list of queues that are located outside
! of our local hub, plus attributes for these associations.
!
! Parameters:
! QUEUENAME     The name of a queue whose messages are to routed outside
!               of the local hub.
! REMOTENODE    A list of nodes to which messages are to be routed from
!               the previously specified queue. At least one node must
!               be in the list for automatic message routing to occur.
!               As many as two nodes may be in the list. If there are
!               no nodes specified, the queue becomes a "POLLED" queue.
!               Remote clients may then poll the queue for messages. The
!               parameter may be either an IP address or a name which
!               can be translated to an IP address using standard socket
!               services.
! REMOTEPORT    The port on which the remote router listens for incoming
!               connect requests. (OPTIONAL) If not specified, the default
!               port number of 3000 is used.
! REMOTEQNAME   The name of the remote queue to which messages from our
```

```
!               queue are routed. (OPTIONAL) Only specified if the name
!               of the remote queue differs from the local QUEUENAME.
! RETRYTIMER    Is the wait period (in minutes) between attempts to
!               connect with a remote node.(OPTIONAL) A default of 1
!               minute is used if the parameter is not specified.
! MINCON        Is another (OPTIONAL) parameter which can specified. It
!               determines how many remote routers must be connected
!               before the local router is allowed to route messages from
!               the local queue.
!
!       Comment delimeters are !
!
[OUTGOING]                                      ! OUTGOING QUEUES
QueueName=REMOTE_QUE1                           ! Queue name
    RemoteNode=PRIMARY_NODE,BACKUP_NODE         ! list of nodes (up to 2 names/addresses)
    RetryTimer=1                                !retry connections (every minute)
    MinCon=1                                    ! Minimum connections required
QueueName=REMOTE_QUE2                           ! Queue name
    RemoteNode=PRIMARY_NODE,BACKUP_NODE         ! List of nodes (Up to 2 names/addresses)
    RetryTimer=1                                !retry connections (every minute)
    MinCon=1                                    ! Minimum connections required
QueueName=REMOTE_QUE3                           ! Queue name
    RemoteNode=PRIMARY_NODE,BACKUP_NODE         ! List of nodes (Up to 2 names/addresses)
    RemoteNode=PRIME_NODE2,BACKUP_NODE2         ! List of nodes (Up to 2 names/addresses)
    RemotePort=2999                             ! Alternate port used at remote router
    RemoteQName=PRIME_QUE3                      ! Name of remote queue to forward to
    RetryTimer=1                                ! retry connections (every minute)
    MinCon=1                                    ! Minimum connections required
```

8.6 TCPIQRSTAT TCP/IP Routing Utility

The TCPIQRSTAT utility is similar in function to the DMPRTR utility except that it returns information pertaining to the TCP/IP IQR Router rather than the DECnet Router. The information is arranged in 4 logical area when output to the user. The first three sections relate directly to the three sections described in the TCP/IP IQR Router Database discussion in the previous chapter.

The top section of the output pertains to the [GLOBAL] parameters for the router. The name of the hub to which the designated router is attached, the port number to which the router is listening for incoming connections, and current date and time.

The next section shows information related to the [INCOMING] section of the router. All incoming queues are listed with the current sequence number for the messages as well as the time of the last write to the queue by the router.

The next section shows information related to the output queues as defined for this router. It lists all the queues which this router will be reading and forwarding to a remote router. Within this section, for each queue, is the time of last read from the queue, the node to which the queue messages are being forwarded, the sequence number of the last message, the state of the connection to the remote router, the length of time that the connection has been established, and the time of the last packet transfer to the remote router.

The last section of the output shows information related to connections which were made to the local router from remote routers. It shows the node which initiated the connection, the sequence number of the last transaction with that router on the connection, the length of time that the connection has been established, and the time at which the last message packet was received from the remote router.

The utility is invoked from the OpenVMS DCL command line with the following syntax:

$ TCPIQRSTAT [hostname] [port]<cr>

If the hostname and port are omitted, it is assumed that the router is located on the local host and is using port 3000 for listening. A connection is made with the router at the designated port for retrieval of router statistics as shown in the following diagram.

9. Utilities

9.1 DMPQUE

This utility is provided to browse the message queues within a hub and to display individual statistics about each of the message queues it contains. The following is displayed for each message queue:

- Time last written
- Time last read
- Elapsed time from read to acknowledge of the last message
- Total elapsed time from read to acknowledge
- VMS process connected for read to the message queue

Calling format:

DMPQUE [/COUNT=*nn*][/SINCE=[*"DD-mmm-yyyy hh:mm:ss"*]] [/MONITOR]
[/FULL]
[/BRIEF] [*hubname*]
[/ACTIVE]

where:

/COUNT	Display only message queues with at least *nn* messages in the message queue.
/SINCE	This will only display message queues that have been read, written, or acknowledged since the provided time stamp. Note that the On_Queue time for a message may still be older than the time specified. By not providing a time stamp, only message queues modified for the current day will be displayed.
/MONITOR	This will activate the monitoring feature of DMPQUE. This will provide a continuously updated display of the hub's message queue information every three seconds. Press CTRL-Y to cancel the display.
/FULL	This causes DMPQUE to display all message queues for the hub, to include deleted message queues. Normally, deleted message queues are not displayed.
/BRIEF	This will cause the output to display only 80 columns of data. Because of the smaller screen size, some of the queue information will not be shown.
/ACTIVE	Displays only those queues that are active (i.e. times are not zero).
hubname	Name of the hub from which to display messages. If not given, DMPQUE will attempt to translate the GROUP or SYSTEM definition of IQRHUB for the default hub name.

The following is a sample screen dump of the called routine:

```
$dmpque/full

HUB information for hub name: TEST_IQR on IPCALP::
HUB Operational since 27-JUL-1995 11:35:44.95  Up for    3 07:00:36.49

Location     Size    Free Blk  Write Cntr  Read Cntr  Act Queues        Last Update
---------  --------  --------  ----------  ---------  ----------  ----------------------
Container      4038      4030          16         28           3  27-JUL-1995 12:51:37.34
Region          197       128          41         41           6  27-JUL-1995 12:31:05.15

Queue Name      Flags   CurMsg MaxMsg LostCnt Last Wrt Last Rd  Last Ack Last Trans CumTran  Primary Reader  SecondaryReader
--------------- -------- ------ ------ ------- -------- -------- -------- ---------- ------- ---------------- ---------------
REPL            ..R..... Replicate to: TEST, MSG_IN
TEST            ......2.      0     20       0 12:29:53 12:31:05 12:31:05    1:11.48 0:08:05
MIKE            ...TW...      0      5       0 00:00:00 00:00:00 00:00:00    0:00.00 0:00:00 IQR_ROUTER
TRACY           ......2.      0     10       0 00:00:00 00:00:00 00:00:00    0:00.00 0:00:00
MESSAGES        .....A2.      0     25       0 10:50:36 10:50:45 10:50:54    0:09.04 0:00:19 IQR_3            IQR_2
MSG_IN          .J....2.      0     20       0 12:51:06 12:51:13 12:51:14    0:07.38 0:02:45
IPACT           XJ....2.      0     20       0 00:00:00 00:00:00 00:00:00    0:00.00 0:00:00
EARL            .J.T..2V      3     10       3 11:20:27 11:20:34 11:20:58    0:23.42 0:00:24 IQR_1
ROUTER          .J..W...      2     10       0 11:31:55 11:32:04 11:32:05    0:00.33 0:00:04 IQR_ROUTER
```

The first line shows the name of the hub and the location of the container file. The next line will show the time the hub was installed in addition to how long the hub has been operational. The top portion of the next display area shows general information about the hub for both its journaled and non-journaled space. The given information is as follows:

Location This will either be “Container” or “Region”. This will indicate what information on the current line is given. Container information is for journaled message queues and Region information is for non-journaled message queues.

Size This is the size of the area in 512 byte blocks.

Free Blk The number of free blocks.

Write Cntr The number of writes made to this particular area.

Read Cntr The number of reads made to this particular area.

Act Queues The total number of active message queues in this area. Deleted message queues are not included in the count.

Last Update The last time the area was updated with information.

The bottom portion of the window shows information about individual message queues on the hub. Given information is as follows:

Queue Name This is the name of the message queue.

Flags This will show current status flags for the message queue. A flag is shown when it is active; otherwise a dot is displayed. Valid codes are as follows:

- **X** Deleted
- **J** Journaled message queue
- **R** Replicating message queue
- **T** Timed message queue (deletes stale messages)
- **W** A reader must be connected in order to write to the queue
- **A** Automatically acknowledge a message read from the queue
- **2** Dual readers allowed (primary and secondary)
- **V** Volatile message queue

CurMsg The current number of messages in the queue that are waiting to be read.

MaxMsg The maximum number of messages that can be in the queue waiting to be read.

LostCnt The number of messages that were deleted in order to make room for new messages (volatile message queue).

Last Write The time of the last write to the queue.

Last Read The time of the last read from the queue.

Last Ack The time of the last acknowledge of a message on the queue.

Last Trans Time (in seconds) that it took between writing a message to the queue and then acknowledging the message.

CumTran The cumulative time that it took between writing a message to the queue and then acknowledging the message.

Primary Reader The process name of the currently connected primary reader.

Secondary Reader The process name of the currently connected secondary reader.

Replicating message queues will not have the usual information found in a regular message queue. Instead, it will list the name of the message queues it will be sending messages to.

9.2 DMPRTR

This utility displays statistics about an IQR router logical link connection status. The format for this utility is as follows:

DMPRTR [*router*]

where *router* is an optionally provided name of a currently running router. If not given, then DMPRTR will default to the name of the router defined by the GROUP or SYSTEM definition of RTRDEF.

The following is a sample display from DMPRTR:

```
$DMPRTR TEST_RTR

Log Node          Flags  DEC Node DECnet Obj  Rem. Link Uptime In Seq# Out Seq#
----------------- ------ -------  ----------- ---------------- ------- --------
MV3               AM      IPCMV3  IPCMV3_RTR     0 00:52:30.49       2       47
ALPHA             DTM    ** No link, retry at: 30-MAY-1995 15:08:05.71 **

                       RTR    Mids  [    Buffers   ]
Object Name          Links  In Out Inp  Out   Resp   RTR Time    RTR Cntr
-------------------- ----- --- --- ---- ----- -----  --------- ------------
    TEST_RTR             2 001 001 0002 00001 00004  14:53:05            10

Size of output buffers:   2048
```

The first section shows the information on all logical links to remote nodes. The information given is as follows:

Log Node The name of a node group defined in the router database.

Flags This can be any of the following:

- **D** Remote link down
- **C** Local connect for remote node in progress
- **L** Local disconnect from remote node in progress
- **Y** Outbound remote link established and connected
- **R** Remote node requesting connection
- **A** Logical link established with remote node
- **T** Connect timer active
- **X** Routing shutdown in progress
- **P** No outbound messages routed by this node
- **W** This node connect race winner
- **O** Remote node is of the old type
- **M** Node supports multipacket messages
- **J** Sending multipacket message
- **N** Negotiate buffer size
- **B** Receiving multipacket message
- **U** Waiting for multipacket size message

DEC Node Actual name of node on the network.

DECnet Obj Name of the router on the DECnet node that is communicatingwith this router.

Rem Link This is elapsed time that the link to the remote node has been up and

Uptime	operational.
In Seq #	Current input sequence number of current node.
Out Seq #	Current output sequence number of current node.

If the remote link is not raised, the *flags* item will contain the status, followed by possible future connection information for the node.

The second part of the display shows information about the local router. Information is as follows:

Object Name	Name of the local router.
RTR Links	Number of links to remote nodes.
MIDS In	The number of Message ID's that are being routed *to* this node.
MIDS Out	The number of Message ID's that are being routed *from* this node.
Buffers Inp	The number of input buffers allocated.
Buffers Out	The number of output buffers allocated.
Buffers Resp	The number of response buffers allocated.
RTR Time	The current time on the local router (adjusted for the network).
RTR Cntr	The number of I/O operations performed by the router.

9.3 DQIT

The DQIT utility provides a simple method of removing/reading messages from a particular message queue within a specific hub. It also has the ability of placing the removed messages into a dump file that can be read by QIT. The command syntax is:

$DQIT /ID=*message_id* [/HUB=*hub_name*] [/TIME] [/NOPRINT] [/COUNT=*nn*] [/WAIT] [/SYMBOL=*symbol*] [/NOACK] [/DUMPFILE=*file*] [/ADD] [/ALL]

where:

/ID=*message_id*	This will indicate the message queue from which messages are to be read. This parameter is required.
/HUB=*hub_name*	This allows the user to specify the name of the particular hub from which the *message_id* is to be found. If not specified, the default specified by the logical IQRHUB will be used.
/TIME	This will take the first 8 bytes of the message and convert them into a VMS equivalent 23 character time using SYS$ASCTIM.
/NOPRINT	Will not print out the message or its header information to the display.
/COUNT=*nn*	Specifying this parameter will instruct DQIT to remove *nn* number of messages from the queue. If there are not at least *nn* messages, then all of the messages in the queue will be read. If a *nn* is set to zero, then all messages will be removed from the queue. The default value of /COUNT is one.
/WAIT	Instructs DQIT to wait for a message to arrive in the message queue if the queue is currently empty. Normally, DQIT will return with an error if the message queue is empty.
/SYMBOL =*symbol*	This will set the DCL *symbol* to the value read by DQIT.
/NOACK	This will not acknowledge the message read from the queue. Use this to just browse the top message in a message queue, without actually removing it from the queue. You cannot use this option with /DUMPFILE or /COUNT.
/DUMPFILE =*file*	Messages read from the queue will be placed in a special dump file named *file*. This file can then be used by QIT to re-populate message queues.
/ADD	This option is only valid with /DUMPFILE. If specified the messages read will be added to the current dumpfile specified by /DUMPFILE. This can be used to create one large dump file with all messages for a particular hub and multiple message queues.
/ALL	This will dump all messages from a message queue. This works the same as setting /COUNT=0.

DQIT has the ability to backup a hub. To do so, use the following command format. You must run this command for *each* message queue you want backed up. Note that DQIT will remove messages from the queue, so it may be a good idea to insure that no one is using the message queues before backing them up. Also, the message will be deleted after performing this operation. To restore the queues to their status before the backup, just use QIT to place the messages back onto the queue. The format is:

DQIT /ID=*message_id* /HUB=*hub_name* /DUMPFILE=*file* /ADD /ALL

9.4 IQU

The IQU utility is responsible for maintaining the IQR hub. It creates, installs, and allows the user to define message queues and their characteristics. The IQU utility is invoked via the command prompt. Its function is to provide communication with the IQR hub process and serve as a maintenance tool for the various global sections.

The following major functions are supported:

- IQU /ADD Create a new message queue on a HUB
- IQU /CREATE Create a new HUB
- IQU /DELETE Delete a message queue on a HUB
- IQU /INFO Show current info about the IQR software
- IQU /INSTALL Install a HUB onto the system
- IQU /MODIFY Modify an existing message queue
- IQU /REMOVE Remove a HUB from the system

9.4.1 IQU /ADD

IQU_ADD creates a message queue based on parameters and qualifiers entered on the command line.

The command used to start this routine and its parameter and qualifiers is as follows:

IQU/ADD=*msg_que* [/loc=*directory*] [/lngmax=*nnnn*] [/descrp=*description*]
[/jrn] [/vol] [/cntmax=*nnnn*] [/noack] [/dual] [/expire=*nnnn*] [/reader]
[/replicate=(*msg_que1*[,...*msg_que4*])] *hub_name*

where:

msg_que	The name of the message queue (max 16 characters).
hub_name	The name of the hub (max 8 characters). If not specified, it will default to the logical IQRHUB.
/loc	Location of hub container file. Default is IQR$QQQQ
/lngmax=*nnnn*	Maximum message size in *nnnn* bytes. Default is 8192 bytes
/descrp=*description*	A description of the message queue (max 80 characters).
/jrn	Messages are journaled. Default is messages non-journaled.
/vol	Messages are volatile.
/cntmax=*nnnn*	The maximum number of messages (*nnnn*) in the message queue at any one time. Default is 20.
/noack	Message acknowledgment not required. Automatically performed upon successful read of message queue.
/dual	Dual readers allowed.

/expire=*nnnn*	Messages will become stale (and deleted) after *nnnn* minutes in the queue.
/reader	A reader is required to write to the queue.
/replicate	Makes this message queue a replicating queue. Enter for (*msg_que1*[,...*msg_que4*]) up four message queues that you want this one to replicate to.

You cannot add a message queue that already exists in a hub.

9.4.2 IQU /CREATE

This command will allow the user to create a new hub based on parameters and qualifiers entered on the command line.

The command used to start this routine and its parameter and qualifiers is as follows:

IQU/CREATE [/fsize=*nnnn*] [/oldh=*filename*] [/dump=*filename*] [/loc=*directory*]
hub_name

where:

hub_name	The name of the hub to create. If not specified, it will default to the logical IQRHUB.
/fsize=*nnnn*	Size of hub container data in *nnnn* 512 byte disk blocks. Default is 12096 blocks.
/loc=*directory*	Location of new hub and QND files. Default is IQR$QQQQ

Use this command to prepare a new mesaging hub on your local node. The new hub will contain no messages or message queues.

Creating a new hub while one is in use will create a new hub file. However, the new hub will not be used until it is installed. The new hub will *not* use any of the current message queue definitions or messages.

Do not use IQU/CREATE after a IQU/REMOVE or a system restart unless you want to completely remove all information from your hub!

9.4.3 IQU /DELETE

This command will delete a message queue from a hub. The format for this command is as follows:

IQU/DELETE=*msg_que* [/loc=*directory*] *hub_name*

where:

msg_que	The name of the message queue (max 16 characters).
hub_name	The name of the hub (max 8 characters). If not specified, it will default to the logical IQRHUB.
/loc	Location of hub container file. Default is IQR$QQQQ

Message queues to be deleted must contain no waiting messages.

Note that message queues are not actually deleted, but marked as being so. They will eventually either be removed during a re-install of the hub or when a new message queue is created over it.

9.4.4 IQU /INFO

Issuing this command will display information about your IQR software installation. The command format is:

IQU/INFO

Information presented will include your IQR serial number, version numbers, and any other possible information about your installation.

9.4.5 IQU /INSTALL

IQU_INSTALL will actually install a hub and prepare it for use by the IQR services.

The command used to start this routine and its parameter and qualifiers is as follows:

IQU/INSTALL [/loc=*directory*] [/msize=*nnnn*] [/cchmaxmqd=*nnnn*]
[/regmaxmqd=*nnnn*] *hub_name*

where:

hub_name	The name of the hub to create. If not specified, it will default to the logical IQRHUB.
/loc=*directory*	Location of hub and QND files
/msize=*nnnn*	Size of hub region data in *nnnn* 512 byte memory blocks
/cchmaxmqd=*nnnn*	Maximum number of cached journal message queues
/regmaxmqd=*nnnn*	Maximum number of non-journal message queues

If the hub experiences an abnormal shutdown (i.e. power failure, system crash) or is rundown using the IQU/REMOVE utility, use IQU/INSTALL to restart the hub. This will preserve only journaled message queues and their respective messages. Non-journaled message queues will be re-created, but their messages will be lost.

9.4.6 IQU /MODIFY

IQU MODIFY allows you to modify a message queue's existing configuration based on parameters and qualifiers entered on the command line.

The command used to start this routine and its parameter and qualifiers is as follows:

IQU/MODIFY=*msg_que* [/loc=*directory*] [/lngmax=*nnnn*] [/descrp=*description*][/vol]
[/cntmax=*nnnn*] [/noack] [/dual] [/expire=*nnnn*] [/reader] *hub_name*

where:

msg_que	The name of the message queue (max 16 characters).
hub_name	The name of the hub (max 8 characters). If not specified, it will default to the logical IQRHUB.
/loc	Location of hub container file.
/lngmax=*nnnn*	Maximum message size in *nnnn* bytes.
/descrp=*description*	A description of the message queue (max 80 characters).

/vol	Messages are volatile.
/cntmax=*nnnn*	The maximum number of messages (*nnnn*) in the message queue at any one time.
/noack	Message acknowledgment not required. Automatically performed upon successful read of message queue.
/dual	Dual readers allowed.
/expire=*nnnn*	Messages will become stale (and deleted) after *nnnn* minutes in the queue.
/reader	A reader is required to write to the queue.

The message queue to modify must already exist on the hub and can not be a replicating message queue. When making modifications, insure that you include all of the switches for all of the options you want -- including those that may already be defined. If you do not define a switch, it will be cleared or reset to its default value.

9.4.7 IQU /REMOVE

IQU_REMOVE marks an existing hub for deletion. When no more processes are connected to the hub, it is removed.

The command used to start this routine and it parameter and qualifiers is:

IQU/REMOVE *hub_name*

where:

hub_name	The name of the hub to remove. If not specified, it will default to the logical IQRHUB.

After a hub is removed, it can again be installed by using IQU/INSTALL. If you wish to create a new, empty hub, use the IQU/CREATE command.

Note: Using this command will stop all message queue activity. If any messages existed in the non-journaled region, they will be deleted. All journaled messages will remain if you re-install the hub.

9.4.8 IQU /RESET

This command will reset the statistical counters for either a hub or a message queue. The command format is:

IQU /RESET[=*msg_que*] [/loc=*directory*] *hub_name*

where:

msg_que	The name of the message queue to reset (max 16 characters). If this is not specified, the hub itslef will be reset.
hub_name	The name of the hub (max 8 characters). If not specified, it will default to the logical IQRHUB.
/loc	Location of hub container file. Default is IQR$QQQQ

If the *msg_que* is specified, then that message queue will have it's transaction counters reset to zero along with all of its timers.

If the *msg_que* is not specified, then the transaction counters for the hub will be reset to zero.

9.5 LSTRTR

This utility displays statistics about all message queues routed by a particular router. The format for this utility is as follows:

LSTRTR [*router*]

where *router* is an optional name of a currently running router. If not given, then LSTRTR will default to the name of the router defined by the GROUP or SYSTEM definition of RTRDEF.

A sample output is shown below:

```
$LSTRTR TEST_RTR

MESSAGE          DESTINATION                TIME
SENT                NODE      COUNTER      DD HH:MM:SS      STATUS
-------          ----------- -------      -----------      ------
MSG_OUT          MV3               42      30 09:50:30
ROUTE            MV3               12      30 10:12:24      AP

MESSAGE             SENDING                 TIME
RECVD                NODE     COUNTER      DD HH:MM:SS      STATUS
-------             -------  -------      -----------      ------
MSG_IN              IPCMV3         1      30 09:53:22
DATA                IPACT         23      30 10:05:11
```

The first portion lists all message queues that are being routed out of this node. The information displayed is as follows:

Message ID	This is the name of the message queue.
Destination Node	This is the name of the group of nodes defined in the routing database where this message queue will be routed.
Counter	The number of messages routed from this message queue.
Time	The last time a message was sent from this message queue.

Status Can be any of the following:

- **F** Remote message id queue is full
- **U** Remote message id is unknown
- **Q** Message id not found in local hub
- **Z** Unable to acknowledge message id
- **S** Packet being sent
- **E** Error reading message id from queue
- **A** Message sent to remote, waiting for ack
- **X** Abort transmission
- **W** Error sending packet
- **P** Sending message as a multipacket
- **W** Multipacket wait
- **H** Multipacket wait
- **N** Destination node is unavailable
- **R** RNA for this message id

The second part lists all the message queues that remote routers will connect to and write on this node. The following information is given:

Message ID This is the name of the message queue.

Sending Node This is the name of the node that will write to this message queue.

Counter The number of messages routed to this message queue.

Time The last time a message was last received by this message queue.

Status Can be any of the following:

- **M** Multipacket message in progress
- **A** Ack being sent
- **Q** Message id was not found in local hub
- **E** Queue write error
- **F** Message queue full error

9.6 QIT

The QIT utility provides a simple method of sending messages to a particular message queue within a specific hub. It also has the ability of re-populating messages onto multiple message queues from a dump file created by DQIT. The command syntax is as follows:

$QIT /ID=*message_id* [/HUB=*hub*] [/TIME] [/DUMPFILE] [/DATAFILE] *data*

where:

/ID=*message_id*	This will indicate the message queue to which the typed message *data* is to be written. If the /DUMPFILE option is specified, then this option is ignored.
/HUB=*hub_name*	This allows the user to specify the name of the particular hub from which the *message_id* is to be found. If not specified, the default specified by the logical IQRHUB will be used.
/TIME	This will convert the time specified by *data* into an 8 byte VMS time and place it onto the indicated message queue.
/DUMPFILE	This will re-populate the hub with messages dumped into the file of file name *data*. The dump file is created with the DQIT utility. If the /ID parameter is specified, all messages in the dump file will be written to the message queue specified by *message_id*, regardless of their actual origin.
/DATAFILE	This will write the message given in the file named *data* to the message queue.
data	Provided without the switch /DATAFILE or /DUMPFILE, this is the message to be written to the message queue.

A popular use of QIT is to restore a backup of messages onto a hub. Before restoring, the hub must already contain the message queues that are in the dump file. To restore a backup, enter at the command line:

QIT /HUB=*hub* /DUMPFILE *filename*

9.7 RTRDBS

The RTRDBS command will compile a router database so it may be used by the router. The router database should normally exist in the IQR$RTR directory. The program will also create a command procedure that can be executed that will start the router using this routing database. The format for the command is as follows:

$RTRDBS ***database***

where *databse* is the name of the routing database to be compiled. See the chapter on the Router for more information about the router databse and use of the RTRDBS utility.

9.8 TCPIQRSTAT

The TCPIQRSTAT utility is similar in function to the DMPRTR utility except that it returns information pertaining to the TCP/IP IQR Router rather than the DECnet Router. The information is arranged in 4 logical area when output to the user. The first three sections relate directly to the three sections described in the TCP/IP IQR Router Database discussion in the previous chapter.

The top section of the output pertains to the [GLOBAL] parameters for the router. The name of the hub to which the designated router is attached, the port number to which the router is listening for incoming connections, and current date and time.

The next section shows information related to the [INCOMING] section of the router. All incoming queues are listed with the current sequence number for the messages as well as the time of the last write to the queue by the router.

The next section shows information related to the output queues as defined for this router. It lists all the queues which this router will be reading and forwarding to a remote router. Within this section, for each queue, is the time of last read from the queue, the node to which the queue messages are being forwarded, the sequence number of the last message, the state of the connection to the remote router, the length of time that the connection has been established, and the time of the last packet transfer to the remote router.

The last section of the output shows information related to connections which were made to the local router from remote routers. It shows the node which initiated the connection, the sequence number of the last transaction with that router on the connection, the length of time that the connection has been established, and the time at which the last message packet was received from the remote router.

The utility is invoked from the OpenVMS DCL command line with the following syntax:

$ TCPIQRSTAT [hostname] [port]<cr>

If the hostname and port are omitted, it is assumed that the router is located on the local host and is using port 3000 for listening. A connection is made with the router at the designated port for retrieval of router statistics as shown in the following output from TCPIQRSTAT.

```
TCPIQR info on 127.0.0.1 Port:3000   Mon Apr 17 12:07:00 2000
HUB:TCPAL3_Q

INPUT QUEUES (4)
QUEUE              SEQ#   LAST WT
```

```
TO_ALPHAQ1             0
TO_ALPHAQ2             0
FROM_ALP               0
TEST_QUE         3731568 12:06:59

OUTPUT QUEUES (0)
QUEUE             LAST RD  TO_NODE       SEQ#  STAT    UP TIME    LAST
PKT

INCOMING CONNECTIONS

IP ADDR           SEQ#     UP TIME     LAST PKT ON QUEUE          MD
IPCMV3::         3731527   6d 19:38:22 12:06:59 TEST_QUE          WT
```

10. Appendix

10.1 IQR Glossary

Term	***Definition***
Acknowledge	When a user reads a message from a message queue, it will need to be acknowledged. Acknowledging a message indicates that the caller has read the message and is done with the message -- it can now be deleted from the message queue.
Container File	This is the actual disk location of the hub's data files.
CMQ	Connected Message Queue definition. This is a segment of memory created by connection to a message queue. When connecting a message queue, this value is returned to the caller. It is then passed on to any routines that will use the connected message queue.
DMPQUE	A utility that displays message queue information about a particular hub.
DMPRTR	A utility that displays information about a currently running router.
DQIT	A utility that will allow the user to remove messages from a message queue.
Hub	This is a named location on a particular node in which actual message queues are contained. Each hub can have a set number of message queues, each holding a set number of messages.
Journaled	A journaled message queue is stored in the disk *container file*. This will allow for message recovery in the event of a shutdown or system crash.
LSTRTR	A utility that gives information about the routed message queues for a particular router.
MAQ	The Manufacturing Automation Queue and Routing software. The IQR Software is compatible (using a patch library) with MAQ v5.3.
Message Header	A portion of a message (of size HDR$K_SIZ) that contains information about the message itself.
Message Queue	A queue within a *hub* that contains actual messages. Each message queue can contain a set number of messages.
MQD	The Manufacturing Automation Disk Based Queuer and Router Services. The IQR Software is compatible (using a patch library) with MQD v4.0
Non-journaled	A type of message queue. A non-journaled message queue only exists within memory on a local node. Messages in a non-journaled message queue can be lost after a system crash or shutdown.
PEX	Process Expanded Region. An area mapped in memory when the user attaches to a particular *hub*.

QIT A utility that will allow the user to place messages onto a message queue.

queue_index A special argument used in many of the IQR System Service calls. This value contains an index into the user's *PEX* that holds information about the currently connected message queue.

Replicate A message queue type that will replicate a message written to it onto other defined local message queues. No messages are actually written to a replicating message queue.

Router A program that will move messages to/from a message queue on the current node to/from a remote node's message queue. The remote router can be of the IQR, MAQ, or MQD type.

Router Database A data file written by the user and compiled using the *RTRDBS* utility. This file contains the information about the nodes and message queues to be routed by the IQR Router.

RTRDBS A utility that compiles the *Router Database*.

Stale A message queue type that indicates that messages can become stale after a certain amount of time. When a stale message is found (one that has existed on the queue longer than its allotted time), it is deleted from the queue. Deleted messages cannot be recovered.

Volatile A message queue type that indicates messages can be volatile. A volatile message is one that may be deleted if no more room exists to write a new message. Deleted messages cannot be recovered. If the queue is not volatile and the message queue is full, then an error is returned to the caller.

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

O

P

Q

R

S

T

U

V

W

INDEX