Revision 3.0 March 1995

SOFTWARE USER'S MANUAL

FOR

KQML

(Knowledge Query and

Manipulation Language)

CONTRACT NO. F30602-93-C-01770

CDRL SEQUENCE NO. A008

Prepared for:

United States Air Force
Advanced Research Projects Agency (ARPA)

Prepared by:

Unisys Corporation
70 East Swedesford Road
Paoli, PA 19301

Table of Contents

1. Scope 1
1.1 Identification 1
1.2 System Overview 1
1.2.1 What is KQML? 1
1.2.2 Using KQML in End-User Applications 1
1.2.3 Using the TCP/IP API 1
1.3 Document Overview. 2
2. Referenced Documents 3
2.1 Government Documents 3
2.2 Non-Government Documents 3
3. Execution Procedures 4
3.1 The KQML Lisp API 4
3.1.1 Configuring the KQML Architecture 4
3.1.2 Using KQML Services 5
3.1.3 Offering KQML Services 5
3.1.4 TCP/IP Procedures 6
3.2 The KQML C API 7
3.2.1 Integrating agents with C KQML 7
3.2.2 Functions provided by the KRIL 7
3.2.3 Customizable KQML variables 9
3.2.4 KQML environment variables 10
3.2.5 KQML Data Structures 10
3.3 Running a KQML Facilitator 10
3.3.1 Communication with KQML Facilitator using the Lisp API 11
3.3.2 Communication with KQML Facilitator using the C API 11
3.4 The KQML Language 12
4. Errors 13
4.1 Errors - Lisp API 13
4.1.1 TCP/IP Package Errors 13
4.2 Errors - C API 14
4.2.1 Environment variables 14
4.2.2 Error messages 14
5. Notes 15

1. Scope

1.1 Identification

This Software User's Manual (SUM) provides the procedures for utilizing the Common Lisp implementation of *KQML* Version 1.8 and the C implementation of KQML Version 1.95.

1.2 System Overview

1.2.1 What is KQML?

Modern computing systems often involve multiple intergenerating computations/nodes. Distinct, and often autonomous nodes can be viewed as agents performing within the overall system, in response to messages from other nodes. There are several levels at which agent-based systems must agree, at least in their interfaces, in order to successfully interoperate:

- Transport: how agents send and receive messages;
- Language: what the individual messages mean;
- Policy: how agents structure conversations;
- Architecture: how to connect systems in accordance with constituent protocols.

KQML is primarily concerned with the transport and language levels. It is complementary to work on representation languages for domain content, including the ARPA Knowledge Sharing Initiative's Knowledge Interchange Format (KIF). *KQML* has also been used to transmit object-oriented data. *KQML* is a language for programs to use to communicate attitudes about information, such as querying, stating, believing, requiring, achieving, subscribing, and offering. *KQML* is indifferent to the format of the information itself, thus *KQML* expressions will often contain subexpressions in different content languages.

KQML is most useful for communication among agent-based programs, in the sense that the programs are autonomous and asynchronous.

A *KQML* message is called a performative, in that the message is intended to perform some action by virtue of being sent. A substantial number of *KQML* performatives can be found in the Specification of the KQML Agent-Communication Language (http://www.cs.umbc.edu/kqml/kqmlspec.ps).

1.2.2 Using KQML in End-User Applications

Any programs can become *KQML* agents by choosing to accept messages from other programs or choosing to send messages to other programs, or choosing to do both. There is no constraint that a program be either a server or a client. Programs are viewed as agents which are free to initiate communication or respond to communication.

The *KQML* implementation creates additional processes to handle incoming messages asynchronously. The user's program is free to execute code or respond to local events (e.g., input from the user at the console).

1.2.3 Using the TCP/IP API

The *KQML* specification doesn't specify the architecture of the environment it is used in. It is possible to use *KQML* in a TCP/IP network of multiprocessing systems (such as UNIX workstations). But it is also possible to transmit *KQML* expressions over RS-232 lines or even send them via email. The agents sending them do not have to be multitasking; they can be more primitive computing systems (e.g., computers running MS-DOS). However, each implementation has to make certain assumptions about the environment in which they work.

The Unisys C and Lisp implementations are designed to work in Sun ANSI C and Lucid Common Lisp environments respectively and they assume that communication will be via a network which implements UNIX sockets.

The implementation also assumes that an agent called a facilitator will be running on an accessible and well-known host. The facilitator keeps track of which services are available and at which hosts and IP port they can be reached.

The TCP/IP primitives used by both the C and Lisp *KQML* implementations provide:

- A client facility for opening a connection to a remote TCP/IP service.
- A server facility which listens to multiple TCP/IP ports and transfers incoming data to a function associated with the port. It will also monitor open streams and invoke a specified function when new data becomes available.

1.3 Document Overview.

This Software User's Manual explains how *KQML* is implemented in Common Lisp and C and how to use this implementation to add a *KQML* interface to existing Common Lisp and C programs. Topics covered include

1. How to configure your local environment to run *KQML.*
2. How to modify an application program to utilize existing *KQML* services.
3. How to write a *KQML* agent, which is an application program that offers to answer queries and respond to assertions via *KQML.*
4. How to utilize the lower level TCP/IP functions included with *KQML* for other application programming purposes.

2. Referenced Documents

2.1 Government Documents

Software User's Manual for the Common Lisp Implementation of KQML. Contract No. F30602-91-C-0040, CDRL Seq. No. A0005

Software Design Document for the Initial Common Lisp Implementation of the KQML Knowledge Router and Knowledge Router Interface. Contract No. F30602-91-C-0040, CDRL Seq No. A006

2.2 Non-Government Documents

Mediated Information Systems Technology, http://louise.vfl.paramax.com/

Draft Specification of the KQML Agent-Communication Language, Finin, Weber, et al.

Contact Tim Finin, Computer Science, University of Maryland Baltimore County, Baltimore MD for current status. (http://www.cs.umbc.edu/kqml/kqmlspec.ps)

KQML as an Agent Communication Language, Finin, Fritzson, McKay and McEntire. *The Proceedings of the Third International Conference on Information and Knowledge Management*, ACM Press, November 1994. (http://www.cs.umbc.edu/kqml/papers/kqml-acl.ps)

Mediators in the Architecture of Future Information Systems, Wiederhold, *IEEE Computer*, vol. 25, no. 3, March 1992, pages 38-49.

3. Execution Procedures

The following sections provide instructions for setting up an environment for the successful execution of application programs, or agents, that use the C or Lisp implementations of *KQML.*

3.1 The KQML Lisp API

There are several steps which need to be followed to integrate and run *KQML* in a particular environment. These are:

- Identify functions in the local program which can be used to respond to incoming messages. Different function can be identified for each *KQML* performative; these can be broken down, roughly into queries , assertions and general instructions
- Identify places in the local program where it will be useful to send expressions to remote programs independently of any replies generated in response incoming messages. Insert calls to the *KQML* function send-msg in these places.
- Install a *KQML* facilitator in the local environment or identify a facilitator that is available to the local environment.

3.1.1 Configuring the KQML Architecture

The *KQML* specification doesn't specify the architecture of the environment it is used it. It is possible to use *KQML* in a TCP/IP network of multiprocessing systems (such as UNIX workstations). But it is also possible to transmit *KQML* expressions over RS-232 lines or even send them via email. The agents sending them do not have to be multitasking, they can be more primitive computing systems (e.g., computers running MS-DOS). However, each implementation has to make certain assumptions about the environment in which they work.

The Common Lisp implementation is designed to work in Lucid/Sun Common Lisp (Version 4) and it assumes that communication will be via a network which implements UNIX sockets.

The implementation also assumes that a single agent, called a facilitator , will be running on an accessible and well known host. The facilitator keeps track of which agents are available and which at hosts and IP port they can be reached.

3.1.1.1 Integrating Agents with Lisp KQML

To establish a *KQML* environment in your local area network:

- Install the *KQML* facilitator (See section 3.4 below) or identify a facilitator available to the local environment.
- Set the variables *local-facilitator-name* and *local-facilitator-port* in the file krouter.lisp file.
- Set the variable *local-domain* in the file tcp.lisp. (*local-domain* is set to your local internet domain name, e.g., "vfl.paramax.com". It is usually permissible to leave this field empty for purely local communications.)

The variables may be set in a separate file which loads the components of *KQML*, see the file config.lisp for an example of this.

3.1.1.2 Functions

`(start-krouter service-name)`

This function initializes local data structures and starts the *KQML* listener. It also registers the application under the name service-name with the local facilitator.

(stop-krouter)

Removes the applications registration with the facilitator and stops the *KQML* server.

3.1.1.3 Variables

local-facilitator-name
local-facilitator-port

These variables should be initialized to contain the name and port number at which the local facilitator is available. However, if they are left uninitialized the implementation will consult the UNIX environment variables FACILITATOR_HOST and FACILITATOR_PORT for this information.

3.1.2 Using KQML Services

To use *KQML* to query a remote service or to send a message of any kind to a remote system, you need to send a *KQML* message to the remote system. This is accomplished with the functions send-msg and make-msg

3.1.2.1 Functions

(make-msg performative content &rest arguments)

Constructs and returns a *KQML* message with the specified performative and content field and all other fields provided. A sample use is:

```
(send-msg (make-msg 'ask-one "QUERY" :reply-with t :receiver
'facilitator))
```

(msg-field msg key)

Returns the field identified by key in the msg

The performative of the message can be retrieved with

```
(msg-field message :performative).
```

(send-msg msg)

Delivers the msg to the appropriate host. If the msg contain a :reply-with parameter with an appropriate value then send-msg will wait for a reply and return it as the value of the function otherwise it will return NIL. Note: The value will be a message.

3.1.3 Offering KQML Services

To setup an application to respond to incoming *KQML* queries and assertions you only need to define and register functions which can handle the incoming message content.

3.1.3.1 Functions

(define-interface performative language)

Currently the *KQML* router discriminates among, incoming messages only by what their performative is and what language the content of the message is in. Use define-interface instead of defun to define a function whose only argument is bound to the variable message

For example,

```
(define-interface (query prolog)
     (make-msg 'reply
          (pl-solve-all (list (msg-field message :content)))))
```

This expression defines a function which handles incoming queries whose content are Prolog expressions. The function uses msg-field to extract the content of the message and passes it to the function pl-solve-all and then packages the result into a new reply message.

3.1.4 TCP/IP Procedures

This section describes auxiliary routines, contained in a separate Lisp module which provide Common Lisp access to TCP/IP services. The functions support both clients and servers.

3.1.4.1 Functions

`(connect-to-service hostname service &optional quiet)`

Returns a bi-directional lisp stream to the remote host hostname connected to the port identified by service hostname is a string which is a hostname that can be resolved by the local operating system or is a string containing a standard form internet address (e.g., "128.126.7.41"). The service is either a string identifying a named service (i.e. something contained in /etc/services) or an integer identifying a TCP/IP port number.

The function returns NIL if a connection can not be made.

The optional Boolean argument quiet blocks the printing of error messages when a connection can not be made.

`(connect hostname service &optional quiet)`

Just like connect-to-service except that successive calls with the same hostname and service will return the same stream instead of opening a new one, that is, it maintains a simple connection cache.

A second Boolean returned value indicates whether the connection is a newly opened one (FALSE) or a cached one (TRUE).

`(register-service function)`

This function is for use in creating a TCP/IP server. It establishes a listener at the next available TCP/IP port. Connections made to this port will have their streams passed to the function which is provided as the only argument.

As a side effect, the assigned port number is stored on the property list of the function under the symbol portnum

The function provided should accept two arguments: a bi-directional stream and a host name (the name of the client initiating the connection).

`(start-server)`

Actually starts the TCP/IP server. No listeners are active until this function is executed. This function will also call register-service for all functions in the variable *inet-config*.

`(stop-server)`

Stops the listener process and closes all listener sockets.

3.1.4.2 Variables

`*inet-config*`

This variable contains a list of functions which will automatically be registered as listeners when start-server is called.

`*local-domain*`

This variable should be initialized to be the name of the local internet domain (e.g., "vfl.paramax.com" or "ai.rl.af.mil").

3.2 The KQML C API

There are several steps which need to be followed to integrate and run *KQML* in a particular environment. These are:

- Identify functions in the local program which can be used to respond to incoming messages. Different functions can be identified for each *KQML* performative; these can be broken down roughly into queries, assertions and general instructions. Register these functions with the *KQML* facilitator using the **kqml_register** function.
- Identify places in the local program where it will be useful to send expressions to remote programs independently of any replies generated in response to incoming messages. Insert calls to the *KQML* function kqml_send_msg in these places.
- Initialize UNIX environment variables and customize variables in *KQML* header files.
- Install a *KQML* facilitator in the local environment or identify a remote *KQML* facilitator which can be reached from the local environment.
- Start up the facilitator, if necessary, and the application program(s).

3.2.1 Integrating agents with C KQML

Alter your application program's makefile to include the *KQML* interface library (the KRIL) and *KQML* header files. Insert a call to **kqml_initialize** in the application, to be executed at startup.

Once you have identified the points in your program where it can respond to incoming messages, declare the functions at those points to the facilitator by inserting a call to **kqml_register** for each event-handling function. These calls to **kqml_register** should be executed before the call to **kqml_initialize**.

Next, identify the places where messages need to be sent to a remote service. Substitute calls to **kqml_send_msg**, or another of the API functions used to send messages at those points.

Define the three UNIX environment variables required by *KQML*.

Alter the values of the settable parameters in kqml.h, if desired.

Recompile your program, and the facilitator if you have changed values in kqml.h.

3.2.2 Functions provided by the KRIL

A *KQML* interface library is provided for each implementation of *KQML*. The C KRIL defines six functions, which are detailed below.

kqml-initialize *char *application-name*

This function initializes local data structures and starts the *KQML* router. It also registers the application under the symbolic name *application-name* with the local facilitator. An application must call this function before executing any calls to **kqml_send_msg**.

kqml_register *handler_function* *char *performative* *int blocking*

An application calls this function at startup to tell the router what application function(s) to call when a response arrives asynchronously. This will set up an event handler. *handler_function* is the name of the function to be registered. This function should

accept a void pointer to the content of the message and a pointer to a kqml_message structure (see kqml.h for the description of this structure) as its first and second arguments. The value this function returns will be sent back to the sender of the message as the reply. If the incoming message is a blocking message, the value returned should be the actual response. For non-blocking messages, the value should be an acknowledgment of receipt. *performative* is a kqml performative other than reply. *blocking* is a Boolean indicating whether the function will handle blocking or non-blocking messages.

NOTE: Only blocking is currently implemented.

kqml_send_msg *char *perf char *content char *rec_agent int reply kqml_message **reply_msg ...*

This function sends an expression to the appropriate destination, i.e. receiving agent using the content and the *KQML* performative supplied. *perf* is a *KQML* performative to be used as the performative of the *KQML* message that will be sent, *content* is the content message of the *KQML* message that will be sent, and rec_agent is the name of the receiving agent that is the destination of this *KQML* message. The fourth argument, *reply,* should be TRUE if the routine should block until a result is returned, and FALSE if the routine should not block. The fifth argument, *reply_msg,* is the address of a pointer to a kqml_message structure and will contain the reply to the message sent when the call to **kqml_send_msg** completes. Finally, this function allows for other arguments to be passed in, using varargs to handle this. The purpose of these final arguments is to provide the API user the ability to specify additional keyword/value pairs to be included in the *KQML* message sent out. If no additional keyword/value pairs are to be specified, then this argument should be NULL.

kqml_send_msgv *char *perf char *content char *rec_agent int reply kqml_message **reply_msg char *kv[]*

This function identically to the kqml_send_msg function describe above. The single difference is that whereas **kqml_send_msg** allows for additional keyword/value pairs by using varargs, **kqml_send_msgv** allows the user to specify these keyword/value pairs using a character string array. This will be useful to users who will be calling **kqml_send_msgv** without knowing beforehand the number of additional arguments to be passed.

kqml_deliver_msg *char *perf char *content ...*

kqml_deliver_msg is used by an application program which is supporting the monitor and subscribe *KQML* performatives. Messages sent using kqml_deliver_msg will be sent to all appropriate subscribers. The *KQML* KRIL will automatically handle all incoming requests by clients which contain the monitor or subscribe performative. Whenever an application calls **kqml_deliver_msg**, the KRIL will determine which clients are listening for the information in the message and route that message to them.

build_msg *char *perf char *content ...*

The **build_msg** function is used to construct new *KQML* messages. **build_msg** returns a pointer to a freshly allocated kqml_message structure which contains the performative and fields provided. It is not necessary to use this function to construct a message, because kqml_send_msg will construct one for you, but it is provided for programmers

who are extending the KRIL or need to use the message structure for some other purpose. The *perf* argument is a *KQML* performative to be used as the performative of the new *KQML* message. *content* is the content field of the message constructed. Additional keyword/value pairs, for inclusion in the *KQML* message being built, may be included after the *content* argument.

get_kqml_field *char *field_name kqml_message *message*

get_kqml_field returns the value of the specified field, if it exists, and NULL if no such field exists in the message. *field_name* is the name of the field whose value is being requested, and *message* is the *KQML* message that will be searched for that field. *KQML* fields are identified by their associated keywords or by the word *:performative*.

put_kqml_field *char *field_name kqml_message *message char *new_value*

put_kqml_field will modify the current value of the field/keyword if that field already exists, or it will add the keyword/value pair to the *KQML* message, if that field/keyword does not yet exist. *field_name* is the name of the field whose value is being changed or added, *message* is the *KQML* message to be modified with the new value, and *new_value* is the new value of the field/keyword. *KQML* fields are identified by their associated keywords.

For some special queries, a wrapper for **kqml_send_msg** has been implemented which takes care of defining the required arguments. These special functions are defined below.

send_sql_query *char *perf char *sql_query char *rec_agent int blocking ...*

The function **send_sql_query** sends an SQL query to the database wrapper named by *rec_agent*. *perf* is the performative to be used in the *KQML* message sent out. This performative should be one that is acceptable to the database wrapper being used as the receiving agent, which, in most cases, will be ask-one or ask-all. *sql_query* is the database query to be executed by the remote agent/wrapper. *rec_agent* is the symbolic name of the database wrapper, which must be up and running for the query to succeed. This function will block, waiting for an answer if the value of *blocking* is TRUE, and will return immediately otherwise. **send_sql_query** will return a pointer to a db_info structure (see description in Section 3.5), or NULL if the query could not be executed by the remote server or if there was a problem during transmission of the query.

NOTE: Be sure that a *KQML*-speaking database wrapper is running for the database being accessed before calling this function.

3.2.3 Customizable KQML variables

These variables are all found in $KQML_HOME/include/kqml.h.

MAX_KNET_ENTRIES
Used by the facilitator to determine the maximum number of agents that a facilitator will keep in its internal cache.

KQMLPORT_OFFSET
The offset from port 5000 at which the *KQML* facilitator will listen for connections.

3.2.4 KQML environment variables

KQML requires that the following environment variables be defined in order to run correctly.

FACILITATOR_HOST
The host name of the machine where the facilitator is running.

FACILITATOR_PORT
The port number where the facilitator is listening for connections. This number must equal the KQMLPORT_OFFSET as defined in kqml.h + 5000.

KQML_HOME
The complete path to the directory where *KQML* has been installed**.**

In addition, the environment variable below may optionally be used.

KQML_WAIT_TIME
If set, the value of this environment variable will determine the number of seconds that *KQML* **will wait for the** return of a blocking query. If not set, the number of seconds that *KQML* will wait for the return of a query is 30 seconds.

3.2.5 KQML Data Structures

1. **db_info** would be accessed by applications which use the database wrapper, which execute calls to SAS, or which use mediators. It is used to hold query results and result table information. Its definition is in the include file db_info.h.

```
struct                          db_info                 {
        int             num_rows;
        int             num_cols;
        int             *datatype;
        int             *datalen;
        struct  u_tag   **res;
        }
```

2. **u_tag**, used by **db_info**, allows datatypes STRING or INTEGER which are the two possible types returned in a query result. Its definition is in dataconv.h.

```
struct                          u_tag                   {
        enum_t dtype;
        union                                           {
                int                                     ival;
                char                                    *sval;
                    }   u_tag_u;
```

3.3 Running a KQML Facilitator

The *KQML* facilitator is a stand-alone program, written in C, which functions as a simple database of *KQML* agents. It maintains a list of all active *KQML* agents in the local environment and is used by the router to determine the addresses of named agents that are the intended recipients of *KQML* messages.

Running the facilitator is simple. It is simply started in the background on a selected host. You may choose to add its startup instructions to the /etc/rc.local of a host in your environment in order to ensure that it is always running. The script for starting it up is in **$KQML_HOME/start_facilitator**.

Normally, the facilitator listens on port 5500. This can be changed by editing the file kqml.h, changing the symbolic constant **KQMLPORT_OFFSET**, and rebuilding the facilitator. If you do this, change the UNIX environment variable **FACILITATOR_PORT** to the new port number.

Please note that the details of the facilitator's abilities are not useful to the users of *KQML* since the implementation hides the transactions with the facilitator, but following is a brief description, for the C and Lisp implementations, of the *KQML* messages it handles.

3.3.1 Communication with KQML Facilitator using the Lisp API

The facilitator responds to the following *KQML* messages:

`(tell :content "symbolic-name:host:port")`

Register the *KQML* agent symbolic-name as being active on the host named host at TCP/IP port **`port`**.

`(untell :content "symbolic-name:host:port")`

Remove the registration of this agent from the facilitator.

`(ask-one :content "symbolic-name")`

Ask for one entry in the facilitator for an agent with this symbolic-name.

`(ask-all :content "symbolic-name")`

Ask for all entries in the facilitator for an agent with this symbolic-name.

`(ask-all :content "")`

Ask for all entries in the facilitator for any agent.

3.3.2 Communication with KQML Facilitator using the C API

The facilitator responds to the following *KQML* messages:

(**tell** :content "symbolic-name:host:port")

Register the *KQML* agent *symbolic-name* as being active on the host named *host* at TCP/IP port *port*.

(**untell** :content "symbolic-name:host:port")

Remove the registration of this agent from the facilitator.

(**ask-one** :content "symbolic-name")

Ask for one entry in the facilitator for an agent with this symbolic-name.

(**ask-all** :content "symbolic-name")

Ask for all entries in the facilitator for an agent with this symbolic-name.

(**ask-all** :content "")

Ask for all entries in the facilitator for any agent.

3.4 The KQML Language

Full details describing the current state of *KQML* can be found in the Specification of the KQML Agent-Communication Language.

KQML establishes a standard protocol and set of conventions for communication among software agent performatives which describe the type of communication being initiated (e.g., a query , an assertion , a definition , etc.)

Each expression in *KQML* contains a single performative and a list of parameters in the form of keyword/value pairs.

In addition, the *KQML* language is extensible. Though there exists a basic set of performatives (see Figure below), new performatives may be added to the specification when the need arises.

Basic query performatives:

evaluate, ask-if, ask-in, ask-one, ask-all

Multi-response query performatives:

stream-in, stream-all

Response performatives:

reply, sorry

Generic informational performatives:

tell, achieve, cancel, untell, unachieve

Generator performatives:

standby, ready, next, rest, discard, generator

Capability-definition performatives:

advertise, subscribe, monitor, import, export

Networking performatives:

register, unregister, forward, broadcast, route

There are about two dozen reserved performative names which fall into seven basic categories.

4. Errors

4.1 Errors - Lisp API

4.1.1 TCP/IP Package Errors

Many of the errors listed here do not indicate a fatal error. Because the system will retry connection attempts, the program may eventually succeed in spite of problems. These errors can be suppressed by setting the variable tcp::*krouter-suppress-warnings* to t

tcp_to_service: host <name> not found

You are attempting to reach a host whose name can not be resolved by the local name service. If you are running a Domain Name Server (DNS), it can not resolve the name. If you are running a local Network Information Service (NIS) or relying on a local host table (/etc/hosts) then the host name you are trying to reach is not in your local database.

tcp_to_service: unknown service: <service name>

You have attempted to reach a service port on a remote machine using a symbolic service name which is not known. This should not occur when using *KQML* since it only uses numeric port identifiers. If you are using the TCP/IP interface for another purpose, consult with you system administrator to determine why the service name you are using is not known on your system.

tcp_to_service: can't make connection: <message>
tcp_to_service: <message>

This error occurs when the system has properly resolved the host and service identifiers but is unable to make a connection with the remote system. The UNIX <message> at the end may provide additional information on why the connection can not be made. Frequently, the connection isn't made because there is no process at the remote system to accept the connection.

While trying to establish listener for: <function> on port <port>
register-service: can't make socket: <message>

While trying to establish listener for: <function> on port <port>
register-service: can't set socket option: <message>

While trying to establish listener for: <function> on port <port>
register-service: can't bind socket: <message>

These errors occurs when the system is trying to build a *KQML* listener and is unable to properly allocate a UNIX socket from the operating system. The <message> may give an indication of the reason.

While trying to establish listener for: <function> on port <port>
register-service: can't identify local host: <message>

This message usually signifies a problem with the local UNIX operating system.

Server not started.

The system was unable to start a *KQML* listener. Earlier messages are likely to have explained why.

`get_connection: <message>`

The system tried to accept an incoming message, but was unable to. The <message> is a UNIX system message which may explain why.

4.2 Errors - C API

When problems occur in starting up an agent or a facilitator, check the following items. For further information in diagnosing problems, consult the **C-KQML Installation Manual**.

4.2.1 Environment variables

Be sure that the variable KQML_HOST has been properly initialized to the complete pathname of the directory where *KQML* was installed.

Verify that the UNIX environment variables **FACILITATOR_HOST** and **FACILITATOR_PORT** are set. **FACILITATOR_HOST** should contain the name of the host where the facilitator agent is to run, and **FACILITATOR_PORT** should contain the port number to be used for communications. This value should be 5000 plus the number assigned to the variable **KQMLPORT_OFFSET** in **KQML_HOST/**include/kqml.h. The default value in this file is 500, therefore **FACILITATOR_PORT** should be set to 5500 unless you have changed the value of **KQMLPORT_OFFSET**.

4.2.2 Error messages

Many of the errors listed here do not indicate a fatal error. Because the system will retry connection attempts, the program may eventually succeed in spite of problems. These errors are normally written to standard error.

unable to exec router program; error #

This error occurs when the system is unable to initialize the router program for the application. The error number may give an indication of the reason.

unable to verify router is up
timed out waiting for connection
no confirmation message from router

These errors occur when the application is unable to contact the router.

unable to connect; #

This error occurs when a connection to a service cannot be opened. The error number may indicate the reason.

5. Notes

Improvements to *KQML* which are currently under development include the following:

- An improved facilitator including a persistent internal database and the ability to start-up certain agents if that agent is not currently available.
- An improved router including initialization of the router cache with the address information for known agents.
- A new *conversation module* which will insure that the *KQML* communication protocol is adhered to in all transactions among agents.