C H A P T E R **3**

Implementing an External Interface to the Cisco IOS Gatekeeper

This chapter describes how to implement an external interface to the Cisco IOS Gatekeeper and contains the following sections:

Although the Cisco IOS Gatekeeper provides many functions, there might be the occasion when additional function is desired or needed. For example, an organization could require additional authentication functions, need to implement specific policy controls, or want to use Internet call waiting.

The Gatekeeper Transaction Message Protocol (GKTMP) and the gatekeeper application programming interface (API) were developed to allow communication between the Cisco IOS Gatekeeper and an external application.

GKTMP is based on RAS and provides a set of ASCII request/response messages that can be used to exchange information between the Cisco IOS Gatekeeper and the external application over a TCP connection, and through the use of the gatekeeper API.

The gatekeeper API is object code that contains the API functions, which are designed to work with GKTMP. An external application links with the object code and calls the functions as necessary.

Using the GKTMP and the gatekeeper API, organizations can supplement the functions of the Cisco IOS Gatekeeper with their own external application.

How the External Interface Works

As part of its normal function, a gatekeeper receives certain RAS registration, admission, location, resource, and disengage messages from H.323 endpoints. Typically, the gatekeeper processes these messages and responds to the request. However, with the Cisco IOS Gatekeeper, the GKTMP, and the gatekeeper API, you can supplement or offload the processing of the request to an external application.

In general, the process works as follows:

1. You establish triggers for each external application on the Cisco IOS Gatekeeper. These triggers are based on RAS tags and values.

2. When the Cisco IOS Gatekeeper receives a RAS message from an H.323 endpoint, it compares the message to the triggers.

3. If there is a match, the Cisco IOS Gatekeeper repackages the contents of the RAS message and sends it to the appropriate external application.

4. The gatekeeper API decodes the byte stream and creates a usable C structure.

5. The external application processes the data and sends the results back to the gatekeeper API.

6. The gatekeeper API encodes the response message and sends the data to the Cisco IOS Gatekeeper.

7. The Cisco IOS Gatekeeper performs any additional processing, if necessary, and forwards the results to the requesting H.323 endpoint.

The interaction between the Cisco IOS Gatekeeper and the external application is completely transparent to the H.323 endpoint.

Communication between the Cisco IOS Gatekeeper and the external application is over a TCP connection through the gatekeeper API. Multiple Cisco IOS Gatekeepers can be configured on a single Cisco IOS router as logical gatekeepers. The same TCP connection can be used by all the logical Cisco IOS Gatekeepers. The individual Cisco IOS Gatekeepers are identified by their gatekeeper IDs. This ID is included in all messages that the Cisco IOS Gatekeeper sends to the external application and in all responses that the external application sends back to the Cisco IOS Gatekeeper. If there are different external applications running on the same host, messages to the different external applications can also be multiplexed on the same TCP connection. The external applications are identified by their server IDs.

How Gatekeeper Triggers Work

By default, the Cisco IOS Gatekeeper does not forward any RAS messages to any external applications. If an application is interested in receiving certain RAS messages, it must register this interest with the Cisco IOS Gatekeeper. To determine which RAS messages the Cisco IOS Gatekeeper forwards to the external application, you can specify trigger parameters. If the Cisco IOS Gatekeeper receives a message that satisfies the specified trigger conditions, the message is forwarded to the external application.

If multiple trigger conditions are specified in a single registration message, the Cisco IOS Gatekeeper treats the trigger conditions as "OR" conditions. In other words, if a RAS message received by the gatekeeper meets any of the trigger conditions the message is sent to the external application.

Trigger conditions are optional. If the Cisco IOS Gatekeeper receives a registration that contains no trigger conditions, it forwards all messages of the specified RAS message type to the external application.

If the Cisco IOS Gatekeeper has a registration for a RAS message type and receives another registration for the same RAS message from the same external application with the same priority, the Cisco IOS Gatekeeper uses the new registration and discards the previous one. The Cisco IOS Gatekeeper allows registrations for the same RAS message type with the same priority from multiple servers.

To indicate that the external application is no longer interested in a message, it must unregister its interest. The contents of the unregistration message must match that of the corresponding registration message before the trigger can be removed.

A Cisco IOS Gatekeeper can be statically (through a command-line interface) or dynamically (through the gatekeeper API) configured with trigger parameters.

Note Triggers that are statically configured can be removed only through the command-line interface. Likewise, those triggers that are dynamically configured can be removed or modified only through the gatekeeper API.

Statically Configured Triggers

Statically configured triggers are established on the router using Cisco IOS commands. To configure triggers using the Cisco IOS command line, do the following:

Step 1 Access the Cisco IOS Gatekeeper configuration mode. Enter the following command:

```
gatekeeper
```

Step 2 Enter the trigger configuration mode and specify the RAS message type for the trigger. Enter the following command:

```
server trigger {arq | lcf | lrj | lrq | rrq | urq | rai | drq | brq} gkid priority
server-id server-ip_address server-port
```

Step 3 If the trigger is to send qualifying messages on a notification-only basis, enter the following command:

```
info-only
```

Step 4 If you want to limit the qualifying messages based on the destination information, enter the following command:

```
destination-info {e164 | email-id | h323-id} value
```

You can repeat this command to enter multiple destinations. This command cannot be used with an RAI message trigger.

Step 5 If you want to limit the qualifying messages based on the redirect reason, enter the following command:

```
redirect-reason value
```

You can repeat this command to enter multiple redirect reasons. This command cannot be used with an RAI message trigger.

Step 6 If you want to limit the qualifying messages based on the remote extension address, enter the following command:

```
remote-ext-address value
```

You can repeat this command to enter multiple remote extension addresses.

Step 7 If you want to limit the qualifying messages based on the endpoint type, enter the following command:

```
endpoint-type value
```

You can repeat this command to enter multiple endpoint types. This command cannot be used with a DRQ message trigger.

Step 8 If you want to limit the qualifying messages based on the supported prefix, enter the following command:

```
supported-prefix value
```

You can repeat this command to enter multiple supported prefixes. This command cannot be used with a DRQ message trigger.

Step 9 When you have specified all the parameters for this trigger, exit trigger submode by entering the following command:

    exit

Step 10 Repeat steps Step 2 through Step 9 for each trigger that you want to define.

Step 11 If you want to change the server timeout value for triggers, enter the following command:

    timer server timeout value

To remove a trigger, use the **no server trigger** command. To temporarily suspend a trigger, enter the trigger configuration mode, as described in step Step 2 and enter the **shutdown** subcommand.

For more information about the Cisco IOS commands for configuring triggers, see Chapter 6, "GKTMP Command Reference."

Note With statically configured triggers, the gatekeeper initiates the connection to the external application and keeps the connection open for as long as it is running. If the connection is terminated by the external application, the Cisco IOS Gatekeeper periodically attempts to re-establish the connection.

Dynamically Configured Triggers

Dynamically configured triggers are established using the gatekeeper API and the GKTMP trigger registration messages.

1. The external application creates a trigger and sends it to the Cisco IOS Gatekeeper using the WriteRegisterMessage API function. The triggers are sent in the format for trigger registration messages as prescribed by the GKTMP.

2. In response, the Cisco IOS Gatekeeper sends a message back that indicates whether the registration request has been accepted.

You must send a separate registration message for each message type that you want sent to the external application. If you send a registration message that does not contain any trigger definitions, all messages of the specified type are sent to the external application.

Dynamically configured triggers are removed using the WriteUnregisterMessage API function and GKTMP trigger unregistration messages. Again, the response from the Cisco IOS Gatekeeper indicates whether the unregistration request has been accepted.

API Functions

You can use the following API functions to dynamically configure triggers:

- WriteRegisterMessage—Sends a registration message to the Cisco IOS Gatekeeper. This function reads the information in the GK_REGISTER_MSG_TYPE structure and sends the contents to the Cisco IOS Gatekeeper using the gkHandle read from the GKAPI_SOCK_INFO_T structure. The header structure, REGISTER_REQUEST_HEADER_TYPE, within each message structure must

contain information for the From, To, and Priority fields. Optionally, if the external application is interested in receiving only notification of a message (not in processing any data for the message), the notificationOnly field should be set to True. Otherwise, it is set to False.

If no filter conditions are to be sent, the parameters within the registration structure should be set to their initialization values or to NULL for pointers. WriteRegisterMessage processes the filters for sending until it reaches the first initialization value for the parameter, or the first null pointer for pointer types.

- WriteUnregisterMessage—Sends an unregistration message to the Cisco IOS Gatekeeper. This function reads the information in the GK_REGISTER_MSG_TYPE structure and sends the contents to the Cisco IOS Gatekeeper using the gkHandle read from the GKAPI_SOCK_INFO_T structure. The header structure, REGISTER_REQUEST_HEADER_TYPE, within each message structure must contain information for the From, To, and Priority fields.

GKTMP Messages

The format of the GKTMP registration/unregistration request and response messages is as follows:

```
Message line
Message header line 1
Message header line 2
Message header line x

Message body line 1
Message body line 2
Message body line x
```

The request and response messages contain the following fields:

- Message line—A single line indicating whether this message is a REGISTRATION or UNREGISTRATION request from the external application. This line is echoed in the response from the Cisco IOS Gatekeeper. The format is REGISTER *xxx* or UNREGISTER *xxx*.

- Message header—A series of lines indicating the server ID of the external application, the gatekeeper ID of the Cisco IOS Gatekeeper, and the priority of the trigger. The priority indicates the order in which this trigger should be processed with respect to other triggers. The message header also includes a version ID, which indicates the version of the GKTMP. The version ID must be the first header in every GKTMP message.

  For trigger registration requests, if the message contains a body, the header can also contain a line indicating the content length of the body. The message header might also contain a line that indicates whether the external application only wants to be notified of the specified RAS messages that the Cisco IOS Gatekeeper receives. For more information on notification only, see the "Notification-Only Triggers" section on page 3-7.

  For trigger registration and unregistration responses, the header also contains a line that indicates the status of the registration or unregistration request.

  The format of each line is *field*:*value*.

- An empty line.

- Message body (optional)—The body of trigger registration messages contains the RAS tags and values that define the desired triggering parameters. Each triggering parameter occupies a single line. The format of each line is *tag=value*.

  The message body can be included only in trigger registration requests. Trigger registration responses and trigger unregistration requests and responses cannot contain a message body.

For more information about the format of trigger registration and unregistration messages, see Chapter 4, "GKTMP Messages."

With dynamically configured triggers, the external application establishes a TCP connection to the gatekeeper and registers its interest in any of the RAS message types. The external application should then leave the connection open for receiving such messages and for sending its responses. If the external application closes the connection, its registrations are considered cancelled. The Cisco IOS Gatekeeper does not attempt to re-establish the connection.

Example of a Dynamic Trigger Registration Message

In the following example the trigger registration request indicates that the Cisco IOS Gatekeeper should forward to the external application any RRQ messages from a voice gateway or a gateway with a supported prefix of 1# or 2#:

```
REGISTER RRQ
Version-id: 100
From: server-12
To: gk-dallas1
Priority: 20
Notification-Only:
Content-Length: 29

t=voice-gateway
p=1#
p=2#
```

Specifying Wildcards in Triggers

Within a trigger, certain wildcard characters are allowed for an alias-address field that contains an E.164 address. Trigger criteria for E.164 alias-addresses can include trailing wildcard characters as follows:

- One or more periods can be used, each denoting a single character

- An asterisk can be used to denote zero or more characters.

Examples of legal E164 address patterns are:

1800.......	The digits 1800 followed by seven characters.
011*	The digits 011 followed by any number of characters.

Examples of illegal E164 address patterns are:

...4567	Wildcard characters must be used as trailing characters. They cannot be used at the beginning of a field.
4802*2	Wildcard characters cannot be used within a field. In this case, the asterisk is interpreted as a literal character.

Notification-Only Triggers

If the application needs to be aware of messages but will not be preforming any processing of the message, you can indicate that the messages should be forwarded on a notification-only basis. If notification-only is present in a GKTMP registration message (which means that notification-only is set to true at the API), the Cisco IOS Gatekeeper forwards messages that meet the trigger criteria but does not expect a response. If notification-only is not present (which means that notification-only is set to false at the API), the Cisco IOS Gatekeeper forwards messages that meet the trigger criteria and awaits a corresponding RESPONSE message from the external application.

This header line is typically used for REQUEST RRQ and REQUEST URQ messages, so that the Cisco IOS Gatekeeper can populate an external application's registration database.

How RAS Messages are Processed

When the Cisco IOS Gatekeeper receives an RAS message that meets the specified trigger conditions, it packages the contents of the fields of the RAS message into the message body of a GKTMP REQUEST message. When the external application receives a request, it must package the response into the message body of a GKTMP RESPONSE message.

The GKTMP specifies formats for exchanging the following types of RAS messages:

- RRQ—Registration request
- RCF—Registration confirm
- RRJ—Registration reject
- URQ—Unregistration request
- ARQ—Admission request
- ACF—Admission confirm
- ARJ—Admission reject
- LRQ—Location request
- LCF—Location confirm
- LRJ—Location reject
- RIP—Request in progress
- RAI—Resource availability indication
- DRQ—Disengage request
- BRQ—Bandwidth request
- BCF—Bandwidth confirm
- BRJ—Bandwidth reject

The URQ, RAI, and DRQ messages are issued as a request from the Cisco IOS Gatekeeper, but do not have a corresponding response. Other messages (RCF, RRJ, ACF, ARJ, BCF, and BRJ) are sent only as responses from the external application.

Note   The Cisco IOS Gatekeeper does not generate GKTMP Request RRQ messages for lightweight RRQ messages, which are used by H.323 endpoints as a keep-alive mechanism to refresh existing registrations.

The general format of the GKTMP RAS messages is:

```
Message line
Message header line 1
Message header line 2
Message header line x

Message body line 1
Message body line 2
Message body line x
```

These messages include the following fields:

- Message line—A single line indicating whether this message is a REQUEST or RESPONSE. The format is REQUEST *xxx* or RESPONSE *xxx*.

- Message header—A series of lines indicating the server ID of the external application and the gatekeeper ID of the Cisco IOS Gatekeeper. The message header also includes a version ID, which indicates the version of the GKTMP. The version ID must be the first header in every GKTMP message.

  If the message contains a body, the header can also contain a line indicating the content length of the body. The header can also contain a transaction ID, which uniquely identifies the request/response message. The message header might also contain a line that indicates whether the message is being sent on a notification-only basis. For more information on notification only, see the "Notification-Only Triggers" section on page 3-7.

  The format of each line is *field*:*value*.

- An empty line.

- Message body (optional)—The body of trigger registration messages contains the RAS tags and values for the corresponding RAS message. The tags included in the body vary depending on the type of RAS message. Responses from the external application should contain only changed or new body information. The format of each line is *tag=value*.

For more information about the format of GKTMP RAS messages, see Chapter 4, "GKTMP Messages."

How the external application processes requests from the Cisco IOS Gatekeeper depends on the type of RAS message and how the external application has been configured to respond.

Note   The Cisco IOS Gatekeeper maintains a timeout value for the processing of requests. If a response is not received within the timeout value, the Cisco IOS Gatekeeper assumes the external application is unavailable. Therefore, when the external application receives a message that will take additional time to process, it should send a message back to the Cisco IOS Gatekeeper to request an extension to the timeout. This message is a RESPONSE RIP.

Processing of *x*RQ Requests

When the external application receives a REQUEST *x*RQ message from the Cisco IOS Gatekeeper it must take one of the following actions:

- Instruct the Cisco IOS Gatekeeper to reject the request. In this case, the external application sends a RESPONSE *x*RJ message to the Cisco IOS Gatekeeper.

- Modify one or more of the fields and return the request to the Cisco IOS Gatekeeper for further processing. In this case, the external application sends a RESPONSE *x*RQ message with the altered information in the body. Only fields that the external application changes can be included in the body. Unchanged fields must not be present in the response message body.

- Indicate no interest in the message and instruct the gatekeeper to continue normal processing. In this case, the external application sends a RESPONSE *x*RQ message with a null message body.

- Complete the processing of the request and send the results to the Cisco IOS Gatekeeper. In this case, the external application sends a RESPONSE *x*CF message. The body of this message must contain all the fields that the Cisco IOS Gatekeeper needs to respond with an *x*CF to the client. This message indicates to the gatekeeper that no further processing is required. If multiple triggers have been configured such that the REQUEST is sent to more than one external application, the RESPONSE *x*CF preempts any other external applications from receiving this message.

- Send no response. This action must be taken only if the request message contains the line Notification-Only: in the header.

Processing of LCF Requests

An LCF message is sent by a peer gatekeeper to confirm the location of a destination endpoint in its zone. You can configure the Cisco IOS Gatekeeper to forward any LCF messages that it receives to the external application. This gives the application an opportunity to alter any of the fields in the confirmation.

When the external application receives a REQUEST LCF message from the Cisco IOS Gatekeeper, the application must take one of the following actions:

- Confirm the information contained in the request. In this case, the external application sends a RESPONSE LCF with a null message body.

- Alter the information contained in the request. In this case, the external application sends a RESPONSE LCF message with the altered information in the message body. Only fields that the external application changes can be included in the body. Unchanged fields must not be present in the response message body.

- Reject the information contained in the LCF. In this case, the external application sends a RESPONSE LRJ.

Processing of LRJ Requests

An LRJ message is sent by a peer gatekeeper to reject the location of a destination endpoint, meaning the endpoint does not exist in the peer gatekeeper's zone. You can configure the Cisco IOS Gatekeeper to forward to the external application any LRJ messages that it receives. This gives the external application an opportunity to recommend an alternate destination.

When the external application receives a REQUEST LRJ message from the Cisco IOS Gatekeeper it must take one of the following actions:

- Accept the LRJ. In this case, the external application sends a RESPONSE LRJ with a null message body.

- Suggest an alternate destination. In this case, the external application sends a RESPONSE LCF message with the altered information in the message body. Only fields that the external application changes can be included in the body. Unchanged fields must not be present in the response message body.

How Security Works

The GKTMP supports the use of CryptoH323Tokens for authentication. The CryptoH323Token is defined in H.225 Version 2 and is used in a "password with hashing" security scheme as described in section 10.3.3 of the H.235 specification.

A cryptoToken can be included in any RAS message and is used to authenticate the sender of the message. The use of cryptoTokens allows you to use a separate database for user ID and password verification.

CryptoTokens and Cisco Gateways

Cisco gateways support the following levels of authentication:

- Registration—Tokens are generated for RRQ and URQ messages.
- Per-Call—Tokens are generated for ARQ messages.
- All—Tokens are generated for RRQ, URQ, and ARQ messages.

You can configure the level of authentication for the gateway using the Cisco IOS software command–line interface.

CryptoTokens for RRQ, URQ, and the terminating side of ARQ messages contain information about the gateway that generated the token, including the gateway ID (which is the H.323 ID configured on the gateway) and the gateway password. CryptoTokens for the originating side ARQ messages contain information about the user that is placing the call, including the user ID and personal identification number (PIN).

Therefore, if you want to use cryptoTokens for authentication, all clients in your network must include a cryptoToken in every message that they send to the Cisco IOS Gatekeeper.

Requirements for using CryptoTokens

To participate in this authentication scheme, a GKTMP-based application must have the following:

- Access to a database of user IDs, gateway IDs, and their associated passwords.
- Access to an ASN.1 encoder.

The application should be set up to authenticate the messages that you deem necessary. If you want to authenticate gateways when they register, your application should validate RRQ messages. If you want per-call authentication, your application should validate ARQ messages. Or, you can have your application validate all messages.

Validating a CryptoToken

To validate a cryptoToken received in a RAS message, the application should:

1. Use the alias in the cryptoToken to look up the associated password.
2. Use the password, the timestamp, and the alias, to ASN.1 encode a ClearToken. The ClearToken is a PwdCertToken. The application should maintain the password and alias as NULL-terminated strings and include the NULL when performing the ASN.1 encoding.
3. Perform an MD5 Hash on the ASN.1 encoded buffer. This results in a 16-byte Hash.
4. Compare the calculated Hash with the one found in the token field of the cryptoEPPwdHash.

If the hash values match, the application should issue a confirmation message (*x*CF) to the gatekeeper, which is transmitted to the gateway. Otherwise, the application should send a rejection message (*x*RJ) with a reject reason of securityDenial.

CryptoTokens in RAS Messages

The cryptoToken message body line contains a type identifier followed by a colon and a sequence of space separated tag=value parameters that are associated with the particular type of cryptoToken.

For example, a message body line containing a cryptoToken could look like the following:

```
$=E:a=H:gw1-rtp T=940647784 h=FFABCD0067AE12436780167364847343
```

For more information about the parameters included in cryptoTokens, see Chapter 4, "GKTMP Messages."

GKTMP Message Examples

The following examples show the GKTMP messages that are generated in some uses of the external interface.

Populating an External Application's Registration Database

An external application might need to maintain a database of active gateways so that it can select gateways for ARQ or LRQ resolution. In this case, triggers can be configured on the Cisco IOS Gatekeepers so that any RRQ or URQ messages will be forwarded to the external application on a notification-only: basis. Example 3-1 shows an RRQ notification for a gateway. Example 3-2 shows a URQ notification for a gateway.

Example 3-1   RRQ Notification

```
REQUEST RRQ
Version-id: 100
From: gk1-sj
Notification-only:
Content-Length:90

c=I:171.69.136.205:1720
r=I:171.69.136.205:16523
a=H:gw3-sj
t=voice-gateway
p=2# 99#
```

Example 3-2   URQ Notification

```
REQUEST URQ
Version-id: 100
From: gk1-sj
Notification-only:
Content-Length:23

c=I:171.69.136.205:1720
```

800 Number Lookup

You might want the Cisco IOS Gatekeeper to forward ARQs to an external application to determine the mapping for an 800 number. Example 3-3 shows an ARQ request from the Cisco IOS Gatekeeper. Example 3-4 shows the corresponding response from the external application.

Example 3-3   Gatekeeper Request

```
REQUEST ARQ
Version-id: 100
From: gk1-sj
Transaction-Id: 5de04245
Content-Length: 127

s=E:4085552132
d=E:8005721234
b=560
A=f
m=t
c=f81d4fae-7dec-11d0-a765-00a0c91e6bf6
C=f81d4fae-7dec-11d0-a765-00a0c91e6bf6
```

Example 3-4   External Application Response

```
RESPONSE ARQ
Version-id: 100
To: gk1-sj
Transaction-Id: 5de04245
Content-Length:14

d=E:4155551212
```

Internet Call-Waiting

If you have an internet call-waiting (ICW) server in your network, you might want configure the Cisco IOS Gatekeeper to forward all LRQ requests to the ICW server. Example 3-5 shows the LRQ request from the Cisco IOS Gatekeeper.

Example 3-5   Gatekeeper LRQ Request

```
REQUEST LRQ
Version-id: 100
From: gk1-sj
Transaction-Id: 5de04246
Content-Length:64

s=H:gk3-la
d=E:4085551111
p=0
c=4085552222
```

If the ICW server determines that the destination (4085551111) is not a subscriber, it sends back a RESPONSE LRQ with a null message body. The Cisco IOS Gatekeeper then proceeds with normal processing. Example 3-6 shows the response.

Example 3-6   Null Response

```
RESPONSE LRQ
Version-id: 100
To: gk1-sj
Transaction-Id: 5de04246
```

If the ICW server determines that the destination (4085551111) is a subscriber and is currently logged on, it pings the subscriber to determine how the call should be handled. Because this can take several seconds, the ICW server first sends a RESPONSE RIP to the Cisco IOS Gatekeeper asking for a 60-second extension to the timeout. Example 3-7 shows the response.

Example 3-7   RIP Response

```
RESPONSE RIP
Version-id: 100
To: gk1-sj
Transaction-Id: 5de04246
Content-Length:7

d=60000
```

If the subscriber refuses the call, the ICW server sends a rejection to the Cisco IOS Gatekeeper. Example 3-8 shows the response.

Example 3-8   Rejection

```
RESPONSE LRJ
Version-id: 100
To: gk1-sj
Transaction-Id: 5de04246
Content-Length:15

R=requestDenied
```

If the subscriber hangs up to accept the call, the ICW server sends a RESPONSE LRQ with a null message body, which instructs the Cisco IOS Gatekeeper to proceed with the call. Example 3-9 shows the response.

Example 3-9   Null Response

```
RESPONSE LRQ
Version-id: 100
To: gk1-sj
Transaction-Id: 5de04246
```

If the subscriber chooses to route the call to voicemail (4085553333) and the ICW server knows the IP address of the voicemail gateway (172.45.63.49), the server instructs the Cisco IOS Gatekeeper to route the call to the voicemail system. Example 3-10 shows the response.

Example 3-10   Response to Reroute

```
RESPONSE LCF
Version-id: 100
To: gk1-sj
Transaction-Id: 5de04246
Content-Length:94
```

```
d=E:4085553333
D=I:172.45.63.49:1720
r=I:172.45.63.49:13982
t=voice-gateway
X=4085551111
```

How the API Works

The gatekeeper API is offered as a library that contains the API functions, which are designed to work with GKTMP. An external application must link with the GKAPI object code and call the API functions to communicate with the Cisco IOS Gatekeeper.

The GKAPI includes the following files:

- gk_api.o—The gatekeeper API object code.

- gk_api.h—The gatekeeper API interface header file, which must be included by the application.

The gatekeeper API provides functions and structures that allow an external application to obtain data from and return information to the Cisco IOS Gatekeeper. Using the API functions and structures, as well as some standard functions, the external client application:

1. Establishes a connection with the Cisco IOS Gatekeeper using the GkapiSetupClient function.

2. Monitors the appropriate socket using a standard function.

3. When a connect complete is detected, the application notifies the gatekeeper API using the GkapiClientConnected function.

4. When a read message is detected, it allocates memory for the storage of the message using the GetReadMsgBuffer function.

5. Stores the contents of the message in the appropriate structure using the ReadMsgBuffer function.

Note   If the message received from the Cisco IOS Gatekeeper is a RAS message that is not supported by the API function, the msgType will be set to MSG_NOT_SUPPORTED. If a response is required, an appropriate response will be constructed by the API function and sent to the Cisco IOS Gatekeeper. The header information in the UNSUPPORTED_MSG_TYPE structure will be filled in by the API function. This situation could occur if the Cisco IOS Gatekeeper has been upgraded to support new messages but the API function has not been correspondingly upgraded.

If the message received from the Cisco IOS Gatekeeper, is not recognized by the API function, the msgType will be set to UNKNOWN_MSG and the STATUS_TYPE will be set to MSG_READ_ERROR. In this case, the external application should close the connection to the Cisco IOS Gatekeeper by calling the CloseGateKeeperConnection function.

If the application has specified the use of non-blocking I/0, the GkapiSetupClient and ReadMsgBuffer functions can return with a CONNECT_IN_PROGRESS or INCOMPLETE_MSG_READ error. These errors indicate that either the connection setup is still in progress or a complete GKTMP message has not been received. If either of these errors is returned, additional socket events will indicate the further processing and completion of these requests. The application should *not* call CloseGateKeeperConnection in these conditions. Instead, the application must monitor the socket using the appropriate handles to detect the additional socket events.

6. Obtains the data from the structure and performs the processing as designed.

7. Frees the memory allocated for the read message using the FreeReadMsgBuffer function.

8. Writes the resulting data to the appropriate structure using the WriteResponseMsg function.

The external application can repeat these steps as often as necessary. If a read error is encountered or if the external application wants to terminate the connection to the Cisco IOS Gatekeeper, the application should use the CloseGateKeeperConnection function.

The API functions and structures are described in Chapter 5, "Gatekeeper API Functions and Structures."

Linking with the Gatekeeper API

As stated earlier, an external application must be linked with the GKAPI object code and call the API functions in order to communicate with the Cisco IOS Gatekeeper. If you have an external application to use with the gatekeeper API and GKTMP, be sure that you link you it with the gatekeeper API library.

The following is an example makefile for building an application using the GNU "C" compiler and linking with the gatekeeper API library.

This sample makefile is bundled with the GKAPI binary and the header file and can be extracted from the GK software .tar file on the Cisco.com website. The sample file can be modified and used to meet the individual requirements of the end user.

```
rm -f gkapiver.c
echo \#ifndef GKAPI_MAX_VER_STR_LEN >>gkapiver.c
echo \#define GKAPI_MAX_VER_STR_LEN 128 >>gkapiver.c
echo \#endif >>gkapiver.c
echo char version_string\[GKAPI_MAX_VER_STR_LEN\]= \"Compiled `date +"%a
%d-%h-%y %H:%M"` OS target: `uname -sr` \"\; >>gkapiver.c
gcc -g -Wall -c gkapiver.c -o gkapiver
gcc -g gk_api gkapiver gk_application.c
/usr/lib/libintl.a -lsocket -lnsl -ldl -lthread
-lpthread -lposix4 -ogk_application
```

Guidelines for Using the Gatekeeper API

When you are writing an application that uses the gatekeeper API, keep the following in mind:

- For response messages, the application must send *only* changed or new parameters. Any unchanged fields must not be included in the response message body. If unchanged fields are sent to the Cisco IOS Gatekeeper in a response message, the performance of the Cisco IOS Gatekeeper could be severely impacted.

- For messages received from the Cisco IOS Gatekeeper, the API function removes the tag fields. The type prefix (H:, E:, M: for alias-addresses and I: for transport-addresses) is preserved and is stored in the appropriate structure. The application must interpret the type of address based on the type prefix.

- For responses from the application, the application must insert the type prefix (H:, E:, M: for alias-addresses and I: for transport-addresses). The API function inserts the appropriate tag before constructing the response message.

- For sequence of parameters in messages received from the Cisco IOS Gatekeeper, the API function removes the tag field and stores the parameter in the appropriate structure in the same format as it was read—with the spaces included in the string.

- For sequence of parameters in responses from the application, the application must separate the parameters with spaces. The API function inserts the appropriate tag before constructing the response message.

- For register functions, "sequence of" parameters are not supported. However, the application can have multiple trigger conditions. This is limited by the maximum size of the array in the registration structures.

- The application is responsible for receiving all signals from the operating system. In order for the API function to detect a closed connection with the gatekeeper during a write operation (so that the STATUS_TYPE can be set to TCP_CONNECTION_CLOSED), the application must install a signal handler for SIGPIPE.

Gatekeeper API Examples

The following examples show how the gatekeeper API functions can be used in an external application. These examples are meant to illustrate how the API functions can be called. They are not examples of actual implementations of the API. Two examples are included in this section; one in which the application is the client and one in which the application is the server.

Example 3-11   Client Example

```
#include "gk_api.h"  /* API header file */
#include </usr/include/sys/fcntl.h>
#include </usr/include/sys/socket.h>
#include </usr/include/sys/select.h>
#include <signal.h>

#define APP_VER 1

void sig_int(int sigNo);
STATUS_TYPE BuildRRQRegisterMsg(GKAPI_SOCK_INFO_T *clientConnect);
STATUS_TYPE BuildRRQResponse(GK_READ_MSG_TYPE *ptr,
                             GKAPI_SOCK_INFO_T *connectPtr);


main()
{
    GKAPI_SOCK_INFO_T clientConnect;
    STATUS_TYPE status;
    GK_READ_MSG_TYPE *readMsgPtr;
    struct timeval tval;
    int conn_handle;
    int n;
    fd_set wset, rset;
    BOOLEAN read_pending = FALSE;

    readMsgPtr=NULL;

    /* Install signal handler for SIGPIPE */
    if (signal(SIGPIPE, sig_int) == SIG_ERR) {
        printf("error registering signal \n");
    }

    /* Open Connection to GateKeeper */
    /* Fill in TCP port and IP address of GateKeeper */
    clientConnect.IPAddress = inet_addr("111.222.111.222");
    clientConnect.TCPPort=2000;
```

```
/* Setup the connection for nonblocking I/O */
conn_handle=GkapiSetupClient(&clientConnect, &status, TRUE);

/* Check status for errors */
/* If status == PROCESSING_SUCCESSFUL, no errors were encountered */
/*    status == TCP_CONNECT_ERROR, error in connecting to GateKeeper */
/*    status == TCP_HANDLE_ERROR, error in handle creation */
/* For error conditions, retry connecting to the GateKeeper */

/* Check for following errors:
 * status = MEM_ALLOC_FAIL
 *          INVALID_MSG_SPECIFIED
 *          INVALID_ENDPOINT_SPECIFIED
 *          INVALID_REDIRECT_REASON_SPECIFIED
 *          HEADER_INFO_INCOMPLETE
 *          NULL_POINTER_PASSED
 */

/* If status is PROCESSING_SUCCESSFUL or CONNECT_IN_PROGRESS,
   wait for connect and read event */
if (status == CONNECT_IN_PROGRESS) {
    FD_ZERO(&rset);
    FD_SET(conn_handle, &rset);
    wset = rset;
    tval.tv_sec = 1;
    tval.tv_usec = 0;
    if ( (n = select(conn_handle + 1, &rset, &wset, NULL,
                    &tval)) == 0) {
        printf("\nApplication connect timed out");
        CloseGateKeeperConnection(&clientConnect);
        exit(1);
    }

    if (FD_ISSET(conn_handle, &rset) ||
        FD_ISSET(conn_handle, &wset)) {
        status = PROCESSING_SUCCESSFUL;
    } else {
        printf("\nSelect error");
        CloseGateKeeperConnection(&clientConnect);
        exit(1);
    }
}

/* If a connect event has occurred tell GKAPI so */
conn_handle = GkapiClientConnected(&clientConnect, &status, conn_handle);

/* If conn_handle is valid and */
/* If status is PROCESSING_SUCCESSFUL, register triggers if required */
/* Build an RRQ Register message */
status = BuildRRQRegisterMsg(&clientConnect);
/* Check status for errors */
/* If status == PROCESSING_SUCCESSFUL, no errors were encountered */
if ((status == TCP_WRITE_ERROR) ||      /* TCP error encountered */
    (status == TCP_CONNECTION_CLOSED)) { /* TCP connection closed */
      /* Close connection to GateKeeper and free system resources */
      CloseGateKeeperConnection(&clientConnect);
}

for(;;) {
  FD_ZERO(&rset);
  FD_SET(conn_handle, &rset);
  select(conn_handle + 1, &rset, NULL, NULL, NULL);
  printf("Select event occurred \n");
  if (FD_ISSET(conn_handle, &rset)) {
```

```
/* If a read event has occurred:
 * Allocate a read buffer
 * Call ReadMsgBuffer
 * Process Message
 * Build Response if required
 */
if (!read_pending)
    readMsgPtr=GetReadMsgBuffer();

/* Check if readMsgPtr is NULL, if NULL, memory allocation failed. */
/* if readMsgPtr != NULL, continue     */
    read_pending = FALSE;
    status=ReadMsgBuffer(&clientConnect, readMsgPtr);



    /* Check status for errors */
    /* If status == PROCESSING_SUCCESSFUL, no errors were encountered */
    if ((status == TCP_READ_ERROR) ||      /* TCP error encountered */
        (status == TCP_CONNECTION_CLOSED) || /* TCP connection closed */
        (status == MSG_READ_ERROR)) {     /* Message not understood */
        /* Free the read buffer
           Close connection to GateKeeper and free system resources
         */
        FreeReadMsgBuffer(readMsgPtr);
        CloseGateKeeperConnection(&clientConnect);
        /* Reopen connection to GateKeeper */
    }

    /* Check for other error conditions:
     * status==MEM_ALLOC_FAIL
     * status==NULL_POINTER_PASSED
     */
        FreeReadMsgBuffer(readMsgPtr);

    /* status==INCOMPLETE_MSG_READ */
    /* Call ReadMsgBuffer on the next read event */
    if (status == INCOMPLETE_MSG_READ)
         read_pending = TRUE;

    /* status == PROCESSING_SUCCESSFUL */
    /* Extract message received        */
    switch(readMsgPtr->msgType) {
        case RRQ_REQUEST_MSG:
            status=BuildRRQResponse(readMsgPtr, &clientConnect);
            /* Check status for errors.
             * If TCP_WRITE_ERROR or TCP_CONNECTION_CLOSED
             * call CloseGateKeeperConnection(&clientConnect)
             * Reopen connection to GateKeeper.
             * Check for other errors.
             */
            FreeReadMsgBuffer(readMsgPtr);
            break;

        case ARQ_REQUEST_MSG:
            /* Do processing */
            FreeReadMsgBuffer(readMsgPtr);
            break;

        case MSG_NOT_SUPPORTED:
            FreeReadMsgBuffer(readMsgPtr);
            break;
```

```
                     default:
                         FreeReadMsgBuffer(readMsgPtr);
                         break;
                 }
             }
         }
}

STATUS_TYPE BuildRRQResponse(GK_READ_MSG_TYPE *ptr,
                             GKAPI_SOCK_INFO_T *connectPtr)
{
    GK_WRITE_MSG_TYPE *writePtr;
    HEADER_INFO_TYPE *headerPtr;
    char buffer1[100];
    char buffer2[100];
    STATUS_TYPE status;

     headerPtr=&ptr->MESSAGE_TYPE.rrqReqMsg.headerInfo;
    /* allocate memory for writePtr, writePtr=malloc(...) */
    /* Fill in msgType and header information */
    writePtr->msgType=RRQ_RESPONSE_MSG;

    writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.headerInfo.versionId = APP_VER;
    strcpy(writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.headerInfo.from,
                      headerPtr->to);
    strcpy(writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.headerInfo.to,
                      headerPtr->from);
    strcpy(writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.headerInfo.
                   transactionID, headerPtr->transactionID);

    /* Fill in paramters */
    strcpy(buffer1, "M:joe_smith");
    strcpy(buffer2, "1800");
    writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.terminalAlias=buffer1;
    writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.supportedPrefix=buffer2;

    /* Send message to GateKeeper */
    status=WriteResponseMsg(connectPtr, writePtr);
    /* If memory was allocated for writePtr, free(writePtr) */
    return(status);
}


STATUS_TYPE BuildRRQRegisterMsg(GKAPI_SOCK_INFO_T *clientConnect)
{
    GK_REGISTER_MSG_TYPE *regPtr;
    STATUS_TYPE status;
    int i=0;
    char buffer1[20];

    /* Allocate memory for regPtr, regPtr=malloc(...) */
    /* After allocating memory:
     * Fill in header info and
     * message parameters if needed
     */

    /* Fill in message type */
    regPtr->msgType = RRQ_REGISTER_MSG;

    /* Fill in header info */
     regPtr->
     REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.versionId =
```

```
                    APP_VER;
            strcpy(regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.from,
                                    "APPL 1");
            strcpy(regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.to,
                                    "GK 1");
            regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.notificationOnly
                                                               =FALSE;
            /* Set priority */
            regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.priority=1;

            /* Specify filters for RRQ message */
            regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.terminalType[0]  =
                                                        VOICEGATEWAY;
            regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.terminalType[1]  = MCU;

            for (i=2; i<MAX_NUM_ENDPOINT_TYPES; i++) {
                regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.terminalType[i]  =
                                                        ENDPOINT_INFO_NOT_RCVD;
            }

            strcpy(buffer1, "1#");
            regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.supportedPrefix[0]=buffer1;
            for (i=1; i< MAX_NUM_SUPPORTED_PREFIX; i++) {
                regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.supportedPrefix[i] = NULL;
            }
            /* Now gatekeeper will only send an RRQ message to the application
             * if filter conditions are satisfied.
             */
            status=WriteRegisterMessage(clientConnect, regPtr);
            /* If memory was allocated for regPtr, free(regPtr) */
            return(status);
}


STATUS_TYPE BuildRRQUnRegisterMsg(GKAPI_SOCK_INFO_T *clientConnect)
{
        GK_UNREGISTER_MSG_TYPE unRegMsg;
        STATUS_TYPE status;

        unRegMsg.versionId = APP_VER;
        strcpy(unRegMsg.from, "APPL 1");
        strcpy(unRegMsg.to, "GK 1");
        unRegMsg.unregisterMsg = RRQ_REGISTER_MSG;
        /* Set priority */
        unRegMsg.priority=1;

        status=WriteUnregisterMessage(clientConnect, &unRegMsg);
        return(status);
}


void sig_int(int sigNo)
{
        switch (sigNo) {
            case SIGPIPE:
                printf("SIGPIPE received\n");
                break;

            /* case ...  */
            default:
        }
}
```

Example 3-12 Server Example

```c
#include "gk_api.h"  /* API header file */
#include </usr/include/sys/socket.h>
#include </usr/include/sys/select.h>
#include </usr/include/netinet/in.h>
#include </usr/include/sys/errno.h>
#include <signal.h>

typedef struct client_db_t_ {
        int handle;
        GK_READ_MSG_TYPE *buf;
        GKAPI_SOCK_INFO_T *conn_info;
} client_db_t;

#define MAX_CLIENTS 1024
#define APP_VER 1

void sig_int(int sigNo);
STATUS_TYPE BuildRRQRegisterMsg(GKAPI_SOCK_INFO_T *clientConnect);
STATUS_TYPE BuildRRQResponse(GK_READ_MSG_TYPE *ptr,
                             GKAPI_SOCK_INFO_T *connectPtr);

main()
{
    GKAPI_SOCK_INFO_T ServerInfo;
    GKAPI_TCP_ADDR_INFO_T client_addr;
    STATUS_TYPE status;
    GK_READ_MSG_TYPE *readMsgPtr;
    GKAPI_SOCK_INFO_T *connInfo;
    client_db_t client[MAX_CLIENTS+1];
    int conn_handle, serverHandle, max_fd;
    int i, n;
    fd_set rset;
    BOOLEAN read_pending = FALSE;

    readMsgPtr=NULL;

    for (i=0; i<=MAX_CLIENTS; i++) {
        client[i].handle=0;
        client[i].buf=0;
        client[i].conn_info=0;
    }

    /* Install signal handler for SIGPIPE */
    if (signal(SIGPIPE, sig_int) == SIG_ERR) {
        printf("error registering signal \n");
    }

    /* Open Connection to GateKeeper */
    /* Fill in TCP port and IP address of Application */
    ServerInfo.IPAddress = inet_addr("111.222.111.222");
    ServerInfo.TCPPort=2000;

    /* Setup the connection for nonblocking I/O */
    serverHandle=GkapiSetupServer(&ServerInfo, &status, TRUE);

    /* Check status for errors */
    /* If the serverHandle < 0, there was an error. Check status for
    /* for the error code.
    /* If status == TCP_CONNECT_ERROR, error in connecting to GateKeeper */
    /*    status == TCP_BIND_ERROR, error in connecting to Gatekeeper */
    /*    status == TCP_LISTEN_ERROR, error in connecting to Gatekeeper */
    /*    status == TCP_NONBLOCK_ERROR, error setting up for  */
```

```
/*                                    nonblocking connection */
/*    status == TCP_HANDLE_ERROR, error in handle creation */
/* For error conditions, quit */
if (serverHandle < 0)
    exit(1);

 /* Set up select mask with the server's handle to listen for
  * incoming connections.
  */

max_fd = serverHandle;

FD_ZERO(&rset);
FD_SET(serverHandle, &rset);

for ( ; ; ) {
    /* If status is PROCESSING_SUCCESSFUL wait for incoming connections
     * and read events */
    n = select(max_fd + 1, &rset, NULL, NULL, NULL);

    /* If the select event has occurred on the server handle,
     * it is a new incoming connection.
     */
    if (FD_ISSET(serverHandle, &rset)) {
        connInfo = (GKAPI_SOCK_INFO_T *)malloc(sizeof(GKAPI_SOCK_INFO_T));
        connInfo->TCPPort = ServerInfo.TCPPort;
        connInfo->IPAddress = ServerInfo.IPAddress;
        conn_handle = GkapiAcceptConnection(connInfo, &status,
                                            serverHandle, &client_addr);
      /* If conn_handle < 0, there is an error. Ignore and continue to
       * to process other select events.
       * If conn_handle is valid, add new connection
       * to select read list
       */
        FD_SET(conn_handle, &rset);

        /* Setup the max file descriptor we need to select on */
        if (conn_handle > max_fd)
            max_fd = conn_handle;

        /* Add this new connection to list of active connections */
        for (i=0; i<MAX_CLIENTS; i++) {
            if (client[i].handle == 0) {
                client[i].handle = conn_handle;
                client[i].buf = 0;
                client[i].conn_info = connInfo;
            }
        }

        /* The application set GK triggers for this connection at
         * this point.
         */
    }

    /*
     * Check to see if the select event is a read occurring
     * on one of the existing connections. If so, have GKAPI process the
     * received buffer.
     */
    for (i=0; n>0,i<=MAX_CLIENTS; i++,n--) {
        if (client[i].handle <= 0)
            continue;

        if (FD_ISSET(client[i].handle, &rset)) {
```

```
                            if (client[i].buf == 0) {
                                readMsgPtr=GetReadMsgBuffer();
                            } else {
                                readMsgPtr=client[i].buf;
                            }

                        /* If a read event has occurred:
                         * Allocate a read buffer if it isn't a pending read.
                         * Call ReadMsgBuffer
                         * Process Message
                         * Build Response if required
                         */
                        if(readMsgPtr != NULL) {
                            status=ReadMsgBuffer(client[i].conn_info, readMsgPtr);

                         /* Check if readMsgPtr is NULL, if NULL,
                          * memory allocation failed.
                          */
                         /* if readMsgPtr != NULL, continue    */
                           if(status == PROCESSING_SUCCESSFUL) {
                               client[i].buf = 0;
                               /* Process the Message */
                               /* Extract message received        */
                               switch(readMsgPtr->msgType) {
                               case RRQ_REQUEST_MSG:
                                   status=BuildRRQResponse(readMsgPtr, &ServerInfo);
                                   /* Check status for errors.
                                    * If TCP_WRITE_ERROR or TCP_CONNECTION_CLOSED
                                    * call CloseGateKeeperConnection(&ServerInfo)
                                    * Reopen connection to GateKeeper.
                                    * Check for other errors.
                                    */
                                   FreeReadMsgBuffer(readMsgPtr);
                                   break;

                                case ARQ_REQUEST_MSG:
                                    /* Do processing */
                                    FreeReadMsgBuffer(readMsgPtr);
                                   break;

                                case MSG_NOT_SUPPORTED:
                                   FreeReadMsgBuffer(readMsgPtr);
                                   break;

                                default:
                                   FreeReadMsgBuffer(readMsgPtr);
                                   break;
                               }

                           }  /* End of status == PROCESSING_SUCCESSFUL */

                           /* Check status for errors */

                           if ((status == TCP_READ_ERROR) ||  /* TCP error encountered */
                               (status == TCP_CONNECTION_CLOSED) || /*connection closed*/
                               (status == MSG_READ_ERROR)) { /* Message not understood */
                                /* Free the read buffer
                                 * Close connection to GateKeeper and
                                 * free system resources
                                 */
                                FreeReadMsgBuffer(readMsgPtr);
                                CloseGateKeeperConnection(client[i].conn_info);
                               /* Reopen connection to GateKeeper */
                           }
```

```
                         /* Check for other error conditions:
                          * status==MEM_ALLOC_FAIL
                          * status==NULL_POINTER_PASSED
                          */
                         FreeReadMsgBuffer(readMsgPtr);

                     /* status==INCOMPLETE_MSG_READ */
                     /* Call ReadMsgBuffer on the next read event */
                     if (status == INCOMPLETE_MSG_READ)
                         client[i].buf = readMsgPtr;


            }
          }
        }
      }
}

STATUS_TYPE BuildRRQResponse(GK_READ_MSG_TYPE *ptr,
                            GKAPI_SOCK_INFO_T *connectPtr)
{
    GK_WRITE_MSG_TYPE *writePtr;
    HEADER_INFO_TYPE *headerPtr;
    char buffer1[100];
    char buffer2[100];
    STATUS_TYPE status;

     headerPtr=&ptr->MESSAGE_TYPE.rrqReqMsg.headerInfo;
    /* allocate memory for writePtr, writePtr=malloc(...) */
    /* Fill in msgType and header information */
    writePtr->msgType=RRQ_RESPONSE_MSG;

    writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.headerInfo.versionId = APP_VER;
    strcpy(writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.headerInfo.from,
                        headerPtr->to);
    strcpy(writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.headerInfo.to,
                        headerPtr->from);
    strcpy(writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.headerInfo.
                    transactionID, headerPtr->transactionID);

    /* Fill in parameters */
    strcpy(buffer1, "M:joe_smith");
    strcpy(buffer2, "1800");
    writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.terminalAlias=buffer1;
    writePtr->WRITE_MESSAGE_TYPE.rrqRespMsg.supportedPrefix=buffer2;

    /* Send message to GateKeeper */
    status=WriteResponseMsg(connectPtr, writePtr);
    /* If memory was allocated for writePtr, free(writePtr) */
    return(status);
}



STATUS_TYPE BuildRRQRegisterMsg(GKAPI_SOCK_INFO_T *ServerInfo)
{
    GK_REGISTER_MSG_TYPE *regPtr;
    STATUS_TYPE status;
    int i=0;
    char buffer1[20];

    /* Allocate memory for regPtr, regPtr=malloc(...) */
    /* After allocating memory:
     * Fill in header info and
```

```
     * message parameters if needed
     */

    /* Fill in message type */
    regPtr->msgType = RRQ_REGISTER_MSG;

    /* Fill in header info */
     regPtr->
     REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.versionId =
         APP_VER;
    strcpy(regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.from,
                            "APPL 1");
    strcpy(regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.to,
                            "GK 1");
    regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.notificationOnly
                                                           =FALSE;
    /* Set priority */
    regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.headerInfo.priority=1;

    /* Specify filters for RRQ message */
    regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.terminalType[0]  =
                                                     VOICEGATEWAY;
    regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.terminalType[1]  = MCU;

    for (i=2; i<MAX_NUM_ENDPOINT_TYPES; i++) {
        regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.terminalType[i]  =
                                              ENDPOINT_INFO_NOT_RCVD;
    }

    strcpy(buffer1, "1#");
    regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.supportedPrefix[0]=buffer1;
    for (i=1; i< MAX_NUM_SUPPORTED_PREFIX; i++) {
        regPtr->REGISTRATION_MESSAGE_TYPE.rrqRegMsg.supportedPrefix[i] = NULL;
    }
    /* Now gatekeeper will only send an RRQ message to the application
     * if filter conditions are satisfied.
     */
    status=WriteRegisterMessage(ServerInfo, regPtr);
    /* If memory was allocated for regPtr, free(regPtr) */
    return(status);
}

STATUS_TYPE BuildRRQUnRegisterMsg(GKAPI_SOCK_INFO_T *ServerInfo)
{
    GK_UNREGISTER_MSG_TYPE unRegMsg;
    STATUS_TYPE status;

    unRegMsg.versionId = APP_VER;
    strcpy(unRegMsg.from, "APPL 1");
    strcpy(unRegMsg.to, "GK 1");
    unRegMsg.unregisterMsg = RRQ_REGISTER_MSG;
    /* Set priority */
    unRegMsg.priority=1;

    status=WriteUnregisterMessage(ServerInfo, &unRegMsg);
    return(status);
}

void sig_int(int sigNo)
{
    switch (sigNo) {
        case SIGPIPE:
            printf("SIGPIPE received\n");
            break;
```

```
                    /* case ... */
                    default:
                }
            }
```