Freescale Semiconductor
Application Note

Document Number: AN3432
Rev. 0, 07/2007

PowerQUICC™ SIP Firewall Traversal
Using Freescale Stateful Rules

by *B. Fong*
DSD Systems Engineering
Freescale Semiconductor, Inc.
Ottawa, ON

Much has been written on the topic of session initiation protocol (SIP) and the problems that it poses to firewalls. Application-aware firewalls need to peer deep into SIP messages beyond the packet headers to make decisions on how best to treat these control messages and the associated voice channels. It is generally difficult to make decisions based on the content at high rates without some form of hardware acceleration. This application note looks at the use of the Freescale regular expression pattern matching engine (PME) along with its stateful rules to enable content processing of SIP messages at gigabit rates.

Established as one of the signaling protocols of choice for voice over IP (VoIP) implementations, SIP is used by IP-enabled telephones to set up calls across a VoIP network infrastructure. It is through SIP that the recipient is located and the voice connection established.

The VoIP architecture uses the notion of out-of-band signaling where signaling information is carried on a channel separate from the actual voice conversation. The signaling channel uses SIP to convey information on the parameters of the desired media connection, including the called and calling addresses to set up the media (for example, voice) channel.

Contents

1 Stateful Rule Overview 2
2 SIP Firewall Traversal Problem 3
3 SIP Stateful Rule Design 7
3.1 SIP Expressions 7
3.2 SIP Stateful Rule 9
4 Stateful Rule Code 12
5 Summary 17
6 Revision History 17

freescale™ semiconductor

When running SIP through a firewall, the network administrator must consider a host of issues, collectively and generically referred to as SIP firewall traversal problems, including how SIP interacts with network address translation (NAT) and how the firewall recognizes newly-established voice channels. This application note focuses on the latter problem of how a firewall discovers which voice channels are legitimately set up so that it can subsequently allow packets on these channels to pass through.

A typical firewall examines packet headers to determine whether to permit or deny packets based on a relatively static set of rules configured by the operator. For example, a rule can allow outbound TCP connections on port 5060 (that is, SIP) to a specific SIP proxy server. With this rule, the firewall observes the TCP 3-way handshake with the SIP proxy server on port 5060 and sets up a pinhole through the firewall to allow SIP control messages returning from the server to pass through. These out-of-band control messages allow the endpoints to negotiate and agree upon the connection parameters to use for the voice-carrying media channel. These media connection parameters are dynamic and cannot be statically configured on the firewall. The firewall must learn about these media channels and open up pinholes as connections are established. If the firewall does not look into and retrieve the media channel parameters from the SIP messages, it cannot set up pinholes for the resulting media traffic, causing voice packets to be dropped.

Solutions to this problem can range from full parsing of the SIP messages, as in a firewall with an embedded SIP proxy, to opening up a range of user datagram protocol (UDP) ports at the risk of compromising security. This application note proposes a technique short of parsing that uses the PowerQUICC™ MPC8572/MPC8572E processor, its pattern matching engine, and in particular the stateful rule engine (SRE), to scan SIP messages and determine the negotiated media channel.

This document starts with an overview of stateful rules, describes the SIP firewall problem in more depth, and follows up with the design and coding of a simplified rule that shows techniques to address this problem. This example illustrates stateful rule concepts and capabilities and demonstrates how stateful rules can be used in conjunction with regular expression pattern matching. The rule in the example is not intended to be exhaustive, but it lays the groundwork for a more complex rule that works within an overall system to create a high performance SIP-aware firewall.

1 Stateful Rule Overview

Taking the step beyond regular expression pattern matching, stateful rules provide the ability for the pattern matching engine to correlate different matches and conduct contextual searches, protocol tracking, multi-pattern matches, and other advanced processing, all without software intervention. A stateful rule is a set of user-defined actions that the pattern matching engine executes when specified pattern match events occur. The actions include changing state, assignments, bitwise operations, comparisons, and relational operations.

While regular expression pattern matching by itself provides a significant offload, the ability to correlate different pattern matches together and report them as a single event dramatically reduces the software post-processing workload. Rather than just informing software when a pattern is matched, stateful rules allow the pattern matching engine to act on pattern match events so that a match is declared to software only when user-specified conditions are met.

For example, a stateful rule can be written to allow the PME to discriminate between matches found in the header portion versus the body portion of the application message. This rule would be useful if the

presence of the pattern in the header portion requires different treatment than the same pattern in the body portion. In general terms, the scope within the data where the specified pattern is of interest may be important. Strings that match the pattern but lie outside the scope may be irrelevant and should be ignored.

This type of contextual search is useful to distinguish matches that require further software processing from those that do not. Often, the application is unconcerned with the match of a single pattern, taking action only after a combination of matches occurs. Without stateful rules, the application software would have to record each match as reported by the hardware, change state, and wait for the hardware to report further matches. With stateful rules, the application software is notified only when the PME detects the requisite set of patterns. Therefore, the application software is required to take action only for a set of matches, postponing the need for the software to be invoked until concrete actions need to be taken.

To address the SIP firewall problem, a stateful rule can be crafted to monitor the protocol exchange, capture media channel information from the messages, and report to software after the media channel is successfully negotiated. The application can then act on this report to open up pinholes for the media flow.

Stateful rules are written using simple instructions that are compiled into hardware-readable form. The PME contains hardwired logic that fetches and executes the compiled instructions. Most instructions take one or two cycles to execute. The instruction set is intuitive, yet intentionally small to minimize the complexity of both the engine and the rule. For more information on stateful rules, refer to the *Pattern Matcher 1.1 Software User's Guide* (PMSOFTUG), which is available at the Freescale web site listed on the back cover of this application note.

2 SIP Firewall Traversal Problem

While a VoIP network can be implemented in a number of ways, the challenge for the firewall remains the same: it[1] needs to inspect and understand the contents of VoIP control messages transiting through it. To illustrate this problem and the subsequent stateful rule, the configuration shown in Figure 1 is used. This figure shows a set of user agent clients (UACs) behind a firewall at an enterprise. For simplicity, other network equipment such as routers and switches are not shown. In this example, the UACs establish TCP[2] connections with the SIP proxy server outside the firewall at a different site. Whether the SIP proxy server is owned by the enterprise or is part of the service provider VoIP infrastructure is of no consequence.

Call requests, or invitations, are created by the UAC and sent to the SIP proxy over the respective TCP connection. The TCP connection is typically established when the UAC registers itself with the SIP registrar which, in this example, is reached through the SIP proxy. Similarly, incoming calls are passed along the same TCP connection from the SIP proxy to the UAC.

1. The term "firewall" is used loosely in this document as a generic term that also includes application layer gateways.
2. TCP is chosen as an example. The PME operates on application-level content and is generally unaware of the underlying transport.

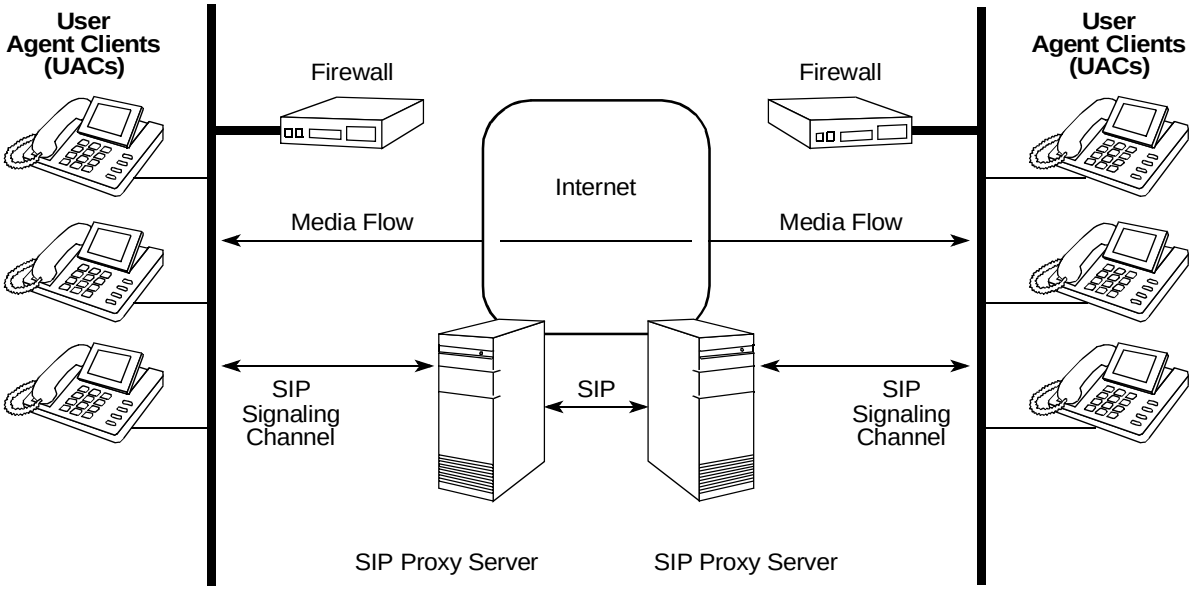

Figure 1. SIP Network

The IP address of the SIP proxy is manually configured—or learned on the UAC through the dynamic host configuration protocol (DHCP) or other means. On startup, the UAC performs the TCP 3-way handshake with the SIP proxy to establish the signaling connection, and registers itself. The registration identifies the UAC to the registrar, and informs the registrar that it is now reachable. It is assumed that the firewall is configured to allow locally-initiated TCP connection requests, either in general or to specific ports such as port 5060, the standard port for SIP messages.

After the UAC registers with the SIP registrar, it can initiate and receive calls. Calls are established through an exchange of messages between the UAC and the proxy server on the signaling channel. SIP proxies, in concert with DNS servers and SIP registrars, have sufficient information to route calls between two registered clients. A successful call setup results in the creation of a media flow that carries the actual voice traffic between the endpoints.

Figure 2 shows the basic call setup procedures. Details, such as how the INVITE message is routed to the destination SIP proxy, are not relevant to the firewall problem and are omitted for clarity. The call setup proceeds as follows:

1. The UAC initiates the call by crafting an INVITE message indicating the desire to connect to the named party.

   The body of the INVITE message contains, among other information, the called party, the calling UAC's IP address, and the port to which the subsequent media flow should be addressed.
2. The SIP proxy processes the INVITE message, returns a TRYING message to the calling UAC, and routes the INVITE message to the called end based on the domain of the called address.
3. The INVITE message reaches the inbound SIP proxy responsible for the called party.
4. The inbound SIP proxy sends the INVITE to the called UAC, which returns a RINGING message to the calling UAC through the proxies.
5. When the called party answers the phone, a 200 OK message is generated and sent back to the calling UAC, again through the proxies.

Within the 200 OK message is the called UAC's IP address and port number.

6. The media channel is established when the calling UAC acknowledges the 200 OK message.

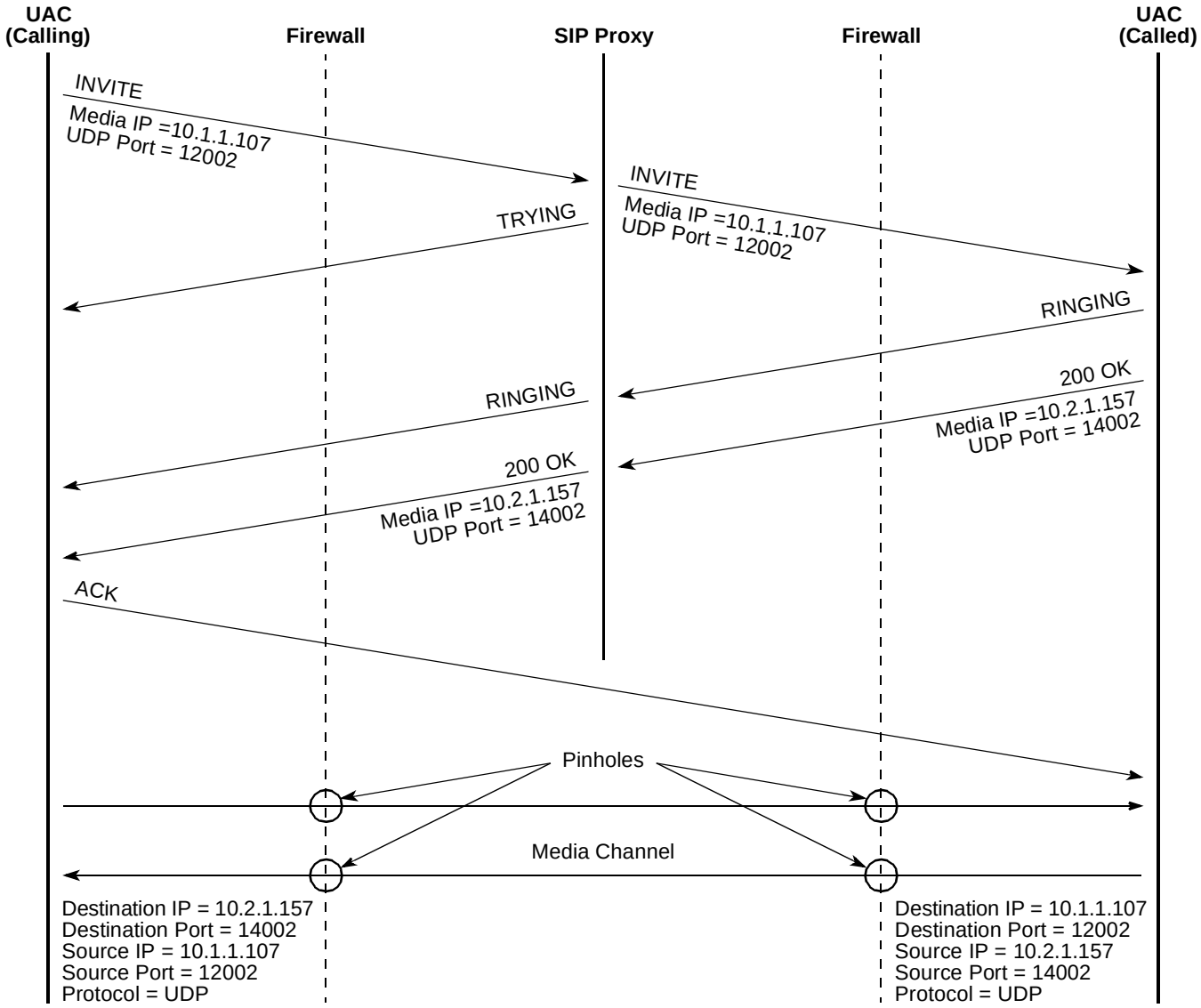

Figure 2. SIP Message Flow

The challenge for the firewall in this example is to look inside the different SIP messages for information on the negotiated media channel and to open up pinholes for subsequent voice packets on the media connection to pass through. SIP is a text-based application protocol in which headers are not fixed to a particular position within the packet, making it impossible to predict with certainty where in the message the media channel parameters can be found.

Example 1 depicts a sample INVITE as it appears to the firewall at the calling end.

Example 1. INVITE Message at the Calling End Firewall

```
INVITE sip:john.doe@freescale.com SIP/2.0
Via: SIP/2.0/TCP camelot-846.am.freescale.com;branch=z9hG4bKdf735Xt
Max-Forwards: 70
To: John <sip:john.doe@freescale.com>
From: Jane <sip:jane.doe@freescale.com>;tag=37462
Call-ID: ae34dae984ff82@freescale.com
CSeq: 1 INVITE
Contact: <sip:jane.doe@freescale.com>
Content-Type: application/sdp
Content-Length: 137

v=0
o=jane 53655765 2353687637 IN IP4 camelot-846.am.freescale.com
s=Session SDP
c=IN IP4 10.1.1.107
t=0 0
m=audio 12002 RTP/AVP 0
```

The INVITE message is divided into two portions, the SIP message portion as specified in RFC 3261 and the session description protocol (SDP) portion as specified in RFC 2327. The beginning of the SDP portion is signified by the `v=` header in the message body. The media connection parameters are in the `c=` and `m=` fields within the SDP portion of the SIP message. The placement (byte offset) of these fields within the message can vary significantly depending on field lengths, optional fields, and whether the parameters are per session or per media. Example 2 shows the 200 OK message as it would appear to the same firewall. Similar to the INVITE message, the media channel parameters in the 200 OK message are specified in the SDP portion, prefaced by the `m=` and `c=` headers.

Example 2. 200 OK Message at the Calling End Firewall

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP camelot-846.am.freescale.com;branch=z9hG4bKdf735Xt;received=10.1.1.107
To: John <sip:john.doe@freescale.com>;tag=738293
From: Jane <sip:jane.doe@freescale.com>;tag=37462
Call-ID: ae34dae984ff82@freescale.com
CSeq: 1 INVITE
Contact: <sip:john.doe@freescale.com>
Content-Type: application/sdp
Content-Length: 131

v=0
o=john 53655766 2353687638 IN IP4 hk-21.ap.freescale.com
s=Session SDP
c=IN IP4 10.2.1.157
t=0 0
m=audio 14002 RTP/AVP 0
```

To determine which pinholes to open for the basic call, the firewall performs the following steps:

1. Recognize that it has received a packet on the SIP signaling channel.

   This step is easily performed because packets on the signaling channel are identified by examining the packet header.
2. Scan the packet looking for specific strings of import:
   a) When an INVITE message is received from the client, look into the SDP portion of the request to obtain the calling IP address and the calling UDP port number for the media channel.
   b) When a 200 OK response to the INVITE message is received from the server, look into the SDP portion of the request to obtain the called IP address and the called UDP port number for the media channel.
3. With the calling and called IP addresses and port numbers, create pinholes to allow packets to flow in both directions on the media channel.

The next section describes how the PME can be used to help with these steps.

3 SIP Stateful Rule Design

The PME consists of a regular expression search engine coupled with a stateful rule engine. The regular expression search engine can be used to match specified strings within the SIP payload, such as to detect the reception of an INVITE message or to locate the connection information. The stateful rule engine can be used to track the protocol exchange by acting on these matches and saving interim state until all negotiated parameters are collected. It is event-driven and reacts only to pattern matches that are defined as events within each rule. Patterns that match but are not defined as events are reported to software in the normal way.

In this example, the requirement is to look for the INVITE message from the calling entity, store the calling IP address and port number, and then look for the 200 OK message from the called entity and store the called IP address and port number. Some validation is performed, but exhaustive checks are left to the application software.

3.1 SIP Expressions

The first step in constructing a stateful rule is to define the expressions that cause this rule to execute. Because the stateful rule performs basic validation of the messages, all the fields that need to be validated are defined as expressions. No assumptions are made about the order of the headers that constitute the SIP message. For a description of the Freescale regular expression syntax, refer to the *Pattern Matcher 1.1 Software User's Guide* (PMSOFTUG).

In Table 1, the expressions are divided into two subsets. Subset 1 represents the expressions to be detected in the calling-to-called direction, and subset 2 represents the expressions to be detected in the called-to-calling direction. In line with the applicable RFCs, all SIP expressions are defined to be case-insensitive, and all SDP expressions are defined to be case-sensitive. Additionally, multi-line is turned on to detect the start of the next header.

Table 1. Expressions for Tracking SIP

Tag (Decimal)	Expression Name	Expression	Case Ins.	Multi-Line	Notes
1	cs_invite	/^INVITE sip:/imtag=0x1 subsets=1	Y	Y	Look for INVITE message.
2	cs_branch	/;branch=z9hG4bK(?$X#[A-Za-z0-9]*)/imtag=0x2 subsets=1	Y	Y	Look for branch parameter and capture it (hashed) into $X.
3	cs_content_sdp	/^CONTENT-TYPE: application\/sdp\s/imtag=0x3 subsets=1	Y	Y	Look for CONTENT-TYPE of sdp.
4	cs_content_other	/^CONTENT-TYPE: application\/([^s]|s[^d]|sd[^p]|sdp\S)/imtag=0x4 subsets=1	Y	Y	Look for CONTENT-TYPE that is not sdp to use as an event that triggers software exception processing.
5	cs_content_length	/^CONTENT-LENGTH: (?$XD[0-9]*)/imtag=0x5 subsets=1	Y	Y	Look for CONTENT-LENGTH header, interpret the subsequent length value as an ASCII-encoded decimal, and capture the resulting number in $X.
6	cs_version	/^v=[0-9]/mtag=0x6 subsets=1	N	Y	Look for the version header that signifies the beginning of the SDP payload.
7	cs_connection1	/^c=IN IP4 (?$XD[0-9]{1,3})\.(?$YD[0-9]{1,3})\./mtag=0x7 subsets=1	N	Y	Look for the connection header (assume IPv4) and store the first two parts of the address into $X and $Y.
8	cs_connection2	/^c=IN IP4 [0-9]{1,3}\.[0-9]{1,3}\.(?$XD[0-9]{1,3})\.(?$YD[0-9])/mtag=0x8 subsets=1	N	Y	Look for the connection header (assume IPv4) and store the second two parts of the address into $X and $Y.
9	cs_media	/^m=[a-zA-Z]+(?$XD[0-9]+).*$/mtag=0x9 subsets=1	N	Y	Look for the media header, interpret the port field as an ASCII-encoded decimal, and store the resulting number into $X.
10	sc_sip_2xx	/^SIP\/2\.0 200 OK/imtag=0xa subsets=2	Y	Y	Look for a SIP 200 OK from the called entity, accepting the invitation to connect.
11	sc_sip_3xx	/^SIP\/2\.0 3[0-9]{2}/imtag=0xb	Y	Y	Look for a SIP 3xx response, indicating a redirection.
12	sc_sip_4xx	/^SIP\/2\.0 4[0-9]{2}/imtag=0xc	Y	Y	Look for a SIP 4xx response, indicating a client error.
13	sc_sip_5xx	/^SIP\/2\.0 5[0-9]{2}/imtag=0xd	Y	Y	Look for a SIP 5xx response, indicating a server error.
14	sc_sip_6xx	/^SIP\/2\.0 6[0-9]{2}/imtag=0xe	Y	Y	Look for a SIP 6xx response, indicating a global error.
15	sc_branch	/;branch=z9hG4bK(?$X#[A-Za-z0-9]*)/imtag=0xf subsets=2	Y	Y	Look for branch parameter and capture it (hashed) into $X.

Table 1. Expressions for Tracking SIP (continued)

Tag (Decimal)	Expression Name	Expression	Case Ins.	Multi-Line	Notes
16	sc_content_sdp	/^CONTENT-TYPE: application\/sdp\s/imtag=0x10 subsets=2	Y	Y	Look for CONTENT-TYPE of sdp.
17	sc_content_other	/^CONTENT-TYPE: application\/([^s]|s[^d]|sd[^p]|sdp\S)/imtag=0x11 subsets=2	Y	Y	Look for CONTENT-TYPE that is not sdp to use as an event that triggers exception processing.
18	sc_content_length	/^CONTENT-LENGTH: (?$XD[0-9]*)/imtag=0x12 subsets=2	Y	Y	Look for CONTENT-LENGTH header, interpret the subsequent length value as an ASCII-encoded decimal, and capture the resulting number in $X.
19	sc_version	/^v=[0-9]/mtag=0x13 subsets=2	N	Y	Look for the version header that signifies the beginning of the SDP payload.
20	sc_connection1	/^c=IN IP4 (?$XD[0-9]{1,3})\.(?$YD[0-9]{1,3})\./mtag=0x14 subsets=2	N	Y	Look for the connection header (assume IPv4) and store the first two parts of the address into $X and $Y.
21	sc_connection2	/^c=IN IP4 [0-9]{1,3}\.[0-9]{1,3}\.(?$XD[0-9]{1,3})\.(?$YD[0-9])/mtag=0x15 subsets=2	N	Y	Look for the connection header (assume IPv4) and store the second two parts of the address into $X and $Y.
22	sc_media	/^m=[a-zA-Z]+ (?$XD[0-9]+).*$/mtag=0x16 subsets=2	N	Y	Look for the media header, interpret the port field as an ASCII-encoded decimal and store the resulting number into $X.

3.2 SIP Stateful Rule

After the expressions are defined, the stateful rule can be written. The SRE is event-driven, so it is critical that the expressions causing this rule to execute be properly and precisely defined.

The locally-originated call state machine for a basic call is illustrated in Figure 3. Starting in RESET_STATE, the state machine progresses as messages from both the client and the server are received and fields detected. The application software is responsible for labeling each direction of this exchange (client-to-server, server-to-client) with a unique stream identifier and for associating these two streams with a single session identifier. The stream identifier allows the PME to correlate consecutive packets within a single flow to match expressions that cross packet boundaries. The session identifier allows the PME to associate the incoming packet with a specific context that describes the current state and state variables for a specific client-server exchange.

This stateful rule is deliberately crafted to highlight capabilities of the SRE and may differ from an actual stateful rule for SIP depending on which fields need to be validated and correlated. For example, this rule reacts only to the INVITE and 200_OK messages and ignores other messages. It also checks the content length, which may or may not apply.

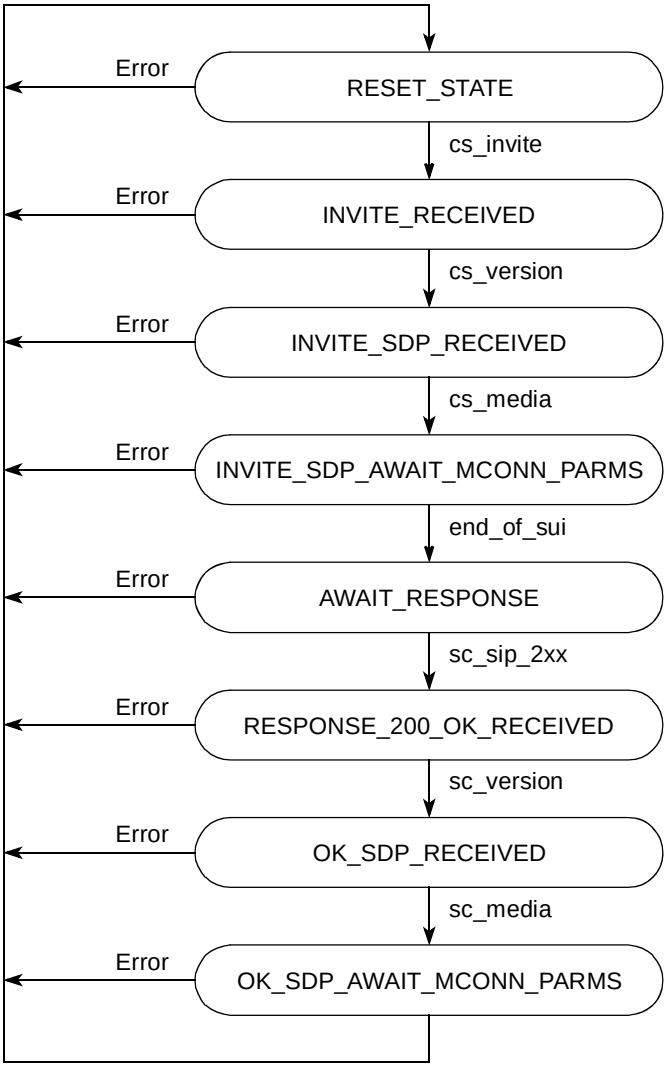

Figure 3. Simplified SIP Protocol State Machine

Table 2 describes the states depicted in Figure 3.

Table 2. Simplified SIP Protocol State Machine

State	Description
RESET_STATE	The initial state of the stateful rule. All variables are automatically initialized to 0. The only expected message in this state is the INVITE message from the client. All other events are ignored. When the INVITE message is received (as detected by matching the *cs_invite* expression), the state is changed to INVITE_RECEIVED.
INVITE_RECEIVED	The call has just been initiated by the local client. In this example, the BRANCH parameter associated with this call is stored for later correlation. This is detected by a positive match with the *cs_branch* expression, which also captures the BRANCH parameter. Other fields such as the FROM or CALL-ID field could also be used. • The CONTENT-TYPE field is checked to ensure that the content is indeed SDP. If not, a report is issued to notify software (which, in a real application, may trigger software parsing of the content). • The CONTENT_LENGTH is detected, captured, and translated into numerical form by the *cs_content_length* expression. This is used later to verify that the SDP portion is indeed bounded by the length specified by this parameter. All of these headers can arrive (that is, be detected) in any order. When the start of the SDP payload is detected (by a positive match with the *cs_version* expression), the state is changed to INVITE_SDP_RECEIVED.
INVITE_SDP_RECEIVED	The start of the SDP portion is detected. The session-level media connection information is detected by positive matches with the *cs_connection1* and *cs_connection2* expressions. These are optional at the session level and may or may not be found. If they are found, the IP address is stored for later reporting. When the media field (*cs_media*) is detected, the port number is translated to numeric form and stored. The state then transitions to INVITE_SDP_AWAIT_MCONN_PARMS.
INVITE_SDP_AWAIT_MCONN_PARMS	Media-level connection information (if present) is detected in this state. After the full SDP portion is received, the state transitions to AWAIT_RESPONSE.
AWAIT_RESPONSE	All relevant information has been gleaned from the local client INVITE message. Now the SIP server response is awaited. This example ignores all SIP responses other than the 200 OK message. When the 200 OK message arrives (as detected by a positive match with the *sc_sip_200_ok* expression), the state is changed to RESPONSE_200_OK_RECEIVED.
RESPONSE_200_OK_RECEIVED	The called client has accepted the call. The BRANCH parameter is verified with the one captured from the INVITE message. When the start of the SDP payload is detected (by a positive match with the *sc_version* expression), the state is changed to OK_SDP_RECEIVED.
OK_SDP_RECEIVED	The start of the SDP portion in the response has been detected. The called IP address (if present) and port number are stored, and the state is changed to OK_SDP_AWAIT_MCONN_PARMS.
OK_SDP_AWAIT_MCONN_PARMS	If media-level connection parameters exist, they are detected and stored. If not, the session-level connection parameters are used. The call is now established and the final acknowledgement from the calling client to the called client is sent directly on the media channel. The call information, consisting of the calling and called IP addresses and port numbers, is reported to the software application.

4 Stateful Rule Code

This section presents code excerpts for the stateful rule described in Section 3, “SIP Stateful Rule Design,” with the goal of highlighting salient features of the Stateful Rule Engine without necessarily going into the details of the rule syntax.

To help with understanding of the code fragments below, a basic explanation of the available registers and variables follows:

- SF[1] through SF[16] are bit-level flags that are available for use. They are persistent for the session, but can be shared by all rules within that session (i.e. per-session variables).
- SRV[1] through SRV[15] are byte-level variables that are available for use. They are persistent within the encompassing rule for the session (i.e. per-rule, per-session variables). To specify multi-byte variables, the notation is SRV[x:y], where x and y are the starting and ending locations respectively. For example, SRV[3:4] refers to a two-byte variable that starts at SRV[3] and ends at SRV[4].
- $X and $Y are capture registers that contain the captured fields as specified in the associated regular expression.
- $P, $NL, $NR are part of a set of registers that allow byte offsets of the current pattern match to be determined.

For a more complete description of stateful rules, refer to the *Pattern Matcher 1.1 Software User’s Guide* (PMSOFTUG).

Initially, the state machine is idle, awaiting the INVITE message whose arrival is indicated by a positive match with the “cs_invite” pattern. All other events are ignored.

```
STATEFUL_RULE: sip

  RESET_STATE:

        EVENT "cs_invite"

                next_state INVITE_RECEIVED
```

Once the INVITE message is detected, the state machine can react to events that represent the various fields of interest within the SIP message. The SIP message consists of a number of mandatory and optional fields followed by media session details contained in the session description protocol (SDP) portion of the content. Since the SIP specification does not mandate a strict order of appearance of the different fields, the stateful rule must be flexible enough to expect fields arriving (i.e. matched) in any order. This can be performed by using separate states to track received events, or, simply as in the framework below, by using session flags (SFs).

```
  STATE INVITE_RECEIVED:

        EVENT "cs_branch"
                SF[1] = 1
        EVENT "cs_content_sdp"
                SF[2] = 1
        EVENT "cs_content_length"
                SF[3] = 1
```

```
EVENT ...
```

All required fields must be present prior to the start of the SDP portion of the INVITE message. The start of the SDP portion is detected by a match with the *cs_version* expression, at which point the code checks to ensure all required fields have been detected prior to changing to the next state. If one or more required fields are missing, the incoming SIP message has been incorrectly formatted, and the event should be reported to software.

```
EVENT "cs_version"
        if (SF[1] == 1) {
                if (SF[2] == 1) {
                        if (SF[3] == 1) {
                                next_state INVITE_SDP_RECEIVED

                                exit

                                }

                        }

                }

        #  Not all expected SIP fields received. Report error to
        #  software.

        report {0x2}
        next_state RESET_STATE
```

The power of the SRE lies in its ability to persistently store fields detected in the data stream for later use. This is shown below in storing the branch parameter and the content length from the SIP message. The session flags (SFs) from before are repeated here for completeness. The stored branch parameter is a hashed value of the branch parameter in the message, and the stored content length is the numeric equivalent of the ASCII-encoded content length. $X is a temporary variable that contains the captured parameter (as specified in the associated regular expression). This captured value needs to be stored into a persistent, per-rule per-session variable or SRV.

```
EVENT "cs_branch"
        SRV[1:3] = $X & 0xffffff
        SF[1] = 1
EVENT "cs_content_length"
        SRV[7:8] = $X
        SF[3] = 1
```

To check for the absence of certain fields, a ‘negative’ expression can be written and detected. For example, the stateful rule only expects SDP-formatted content within the SIP message. If content other than SDP is received, the stateful rule reports this anomaly to software.

```
EVENT "cs_content_other"
        report {0x1}
        next_state RESET_STATE
```

In some situations, it may be useful to check whether certain fields are within a specific part of the message. In SIP, the content length field specifies the length of the SDP portion of the message. In order

to later check whether certain fields are within the SDP portion of the message, three steps need to be performed: i) capture and store the content length (done above), ii) determine the starting offset of the SDP portion and store the expected ending offset (see below), iii) check whether detected fields are within the content length (shown later).

Offsets of a match are accessible through the $P and $NL registers. The sequence below calculates the ending offset of the SDP portion by taking the leftmost offset of the v= field and adding that to the content length stored previously. The v= field signifies the start of the SDP portion of the message and is represented by the *cs_version* expression. At the end of this code fragment, SRV[7:8] will contain the expected ending byte offset of the SDP portion. (Again the session flags have been repeated here for completeness.)

```
        EVENT "cs_version"
                if (SF[1] == 1) {
                        if (SF[2] == 1) {
                                if (SF[3] == 1) {
                                                SRV[7:8] = SRV[7:8] + $P
                                                SRV[7:8] = SRV[7:8] - $NL
                                                next_state INVITE_SDP_RECEIVED

                                                exit

                                                }

                                        }

                                }

                #  Not all expected SIP fields received. Report error to
                #  software.

                report {0x2}
                next_state RESET_STATE
```

With the expected ending offset stored, a simple comparison can be made to ensure each SDP field is within the content length specified. The $P and $NR registers contain the rightmost offsets of the current match.

```
STATE INVITE_SDP_RECEIVED:


        EVENT "cs_connection1"
                SRV[9:10] = $P + $NR
                if (SRV[7:8] <= SRV[9:10]) {
                        #  Connection information field is located outside of
                        #  the range specified by the content length. Report
                        #  error to software.

                        report {0x3 $P}
                        next_state RESET_STATE
                        exit

                    }
```

The SDP portion consists of a session-level section and one or more media sections. In the INVITE message, the fields of interest are the (calling) IP address and the (calling) port number. The IP address

may appear in either the session-level section or the media section or both. If the IP address appears in both, then the IP address in the media section supersedes the address in the session-level section. To handle these possibilities, code (not shown) can store the IP address, if present, from the session-level section, and overwrite this with the address detected in the media section, if present.

In order to store IP addresses, it is necessary to save each part of the dotted quad representation separately. However, since the PME can only capture two fields in any given expression, two expressions need to be written and matched, with the first capturing the first two fields of the dotted quad representation, and the second expression capturing the third and fourth fields. The content length checking code is repeated here for completeness.

```
STATE INVITE_SDP_RECEIVED:

        EVENT "cs_connection1"
                SRV[9:10] = $P + $NR
                if (SRV[7:8] <= SRV[9:10]) {
                        #  Connection information field is located outside of
                        #  the range specified by the content length. Report
                        #  error to software.

                        report {0x3 $P}
                        next_state RESET_STATE
                        exit

                        }
                SRV[9] = $X
                SRV[10] = $Y


        EVENT "cs_connection2"
                #  Store third and fourth parts of the CALLING IP address.
                SRV[11] = $X
                SRV[12] = $Y
```

Similarly, the port number can be extracted and stored. Presence of the port number is indicated by a match with the “cs_media” expression.

```
        EVENT "cs_media"
                SRV[13:14] = $P + $NR
                if (SRV[7:8] <= SRV[13:14]) {
                        #  Media section is located outside of
                        #  the range specified by the content length. Report
                        #  error to software.

                        report {0x4 $P}

                        next_state RESET_STATE
                        exit

                      }
                #  Store the CALLING port number.
                SRV[13:14] = $X
                next_state INVITE_SDP_AWAIT_MCONN_PARMS
```

After the INVITE message is fully processed, with the calling IP address and port number stored in SRV[9] through SRV[14], the rule can progress to a state that looks for server responses.

```
STATE AWAIT_RESPONSE:
         EVENT "sc_sip_2xx"
                  next_state RESPONSE_200_OK_RECEIVED
         EVENT "sc_sip_3xx"
                  ...
         EVENT "sc_sip_4xx"
                  ...
         EVENT "sc_sip_5xx"
                  ...
```

When the 200 OK message is received, the code checks to see if the previously stored branch parameter matches the one in the 200 OK message.

```
STATE RESPONSE_200_OK_RECEIVED:
         EVENT "sc_branch"
                  #  Compare the "branch" parameter in the response with the
                  #  request.
                  SRV[4:6] = $X & 0xffffff
                  if (SRV[4:6] != SRV[1:3]) {
                           # "branch" parameter mismatch. Pass to software.
                           report {0x7 $P}
                           next_state RESET_STATE
                           exit

                           }
```

Processing of the rest of the 200 OK message is similar to that of the INVITE message where the (called) IP address and (called) port numbers are extracted and stored. Once that is performed, the media channel parameters can be reported to software. In the code fragment below, the final part of the IP address has just been detected and captured in $X and $Y. SRV[1:2] contains the first part of the called IP address, SRV[5:6] contains the called port number, and SRV[9:14] contains the calling parameters from the INVITE message.

```
         EVENT "sc_connection2"
                  # CALLED IP address received and captured.
                  report {
                           200
                           SRV[9:14]   # calling info
                           SRV[1:2]    # called info
                           $X:2
                           $Y:2
                           SRV[5:6]
                           }
                  next_state RESET_STATE
                  }
```

5 Summary

This application note focuses on the specific problem of extracting media channel information from the SIP message exchange between a SIP client and a SIP proxy server. The text-based unanchored nature of SIP makes searching and acting on the payload well suited to the capabilities of the Freescale pattern matching engine, which can perform these functions at gigabit rates. To illustrate how pattern matching can be implemented, a set of regular expressions is defined and a simplified stateful rule is written to search through SIP messages looking for and reporting the negotiated media channel parameters to software.

You can use the concepts illustrated in this example to tailor a solution for an overall content-aware firewall application or other application that requires similar functionality.

6 Revision History

Table 3 provides a revision history for this application note.

Table 3. Document Revision History

Rev. Number	Date	Substantive Change(s)
0	07/2007	Initial release.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN3432
Rev. 0
07/2007