Internet Draft
draft-culpepper-sip-key-events-01.txt
March 1, 2002
Expires: September, 2002

Bert Culpepper
InterVoice-Brite, Inc.

Robert Fairlie-Cuninghame
Nuera Communications, Inc.

Jean-Francois Mule
CableLabs

SIP Event Package for Keys

Status of this Memo

Abstract

This document defines a protocol independent message format for requesting and reporting state related to keys or buttons. The document also defines a SIP Event Package to make use of this extensible framework through the SIP SUBSCRIBE & NOTIFY messages. The draft currently defines key event sub-packages to represent the state of telephone dial pads, feature buttons found on many commercial telephones, standard keyboards as well as support for user or device specific buttons (for instance, a "Double Latte" button). The event package and associated key packages can be used to enable new SIP services where an application or SIP entity requires notification of user interface activity either associated

with a particular SIP session or simply in relation to the device itself.

Table of Contents

0. Changes since draft-culpepper-key-events-00

- Substantial reorganization of draft. The draft is now split into two parts: a protocol independent Key-Event Message Format section and a SIP Events specific section. This allows other authors to write transport profiles for other protocols.

- Removed Applies-To header and replaced it will a protocol dependent hole in the application/key-event message body (instance-header).

- Updated to reflect draft-ietf-sip-events-04 changes.

1. Introduction

As stated in SIP-specific Event Notification [2], the SIP SUBSCRIBE and NOTIFY methods provide a mechanism by which cooperating SIP [3] endpoints can request and receive asynchronous notification of changes in state related to network resources and calls. This mechanism, when combined with an event package for keys, can be used for requesting and reporting the status of keys and keypads associated with SIP user agents or indeed any SIP device.

Communication sessions many times are used as a means to access and deliver services to users. The operation and use of these services almost always require some user interaction with the service. One of the primary means for user interactions with services when using the legacy circuit-switched telecommunications network is via tones generated as a result of a key press on the communications device. While tones can be used in the same manner in IP networks, their use, given the distributed nature of the network, is sub-optimal. In addition, their use limits the user interface of many user devices as well as the range of application services possible. It is for these reasons that a means to transport key presses is needed.

2. Motivation/Use Cases

The first and foremost motivation for this proposal is to extend the possibility for user interaction beyond the antiquated telephone dialpad model imposed by limiting interaction to DTMF tones. The proposal is designed to support full keyboards, multiple key instances, user- or device-specific buttons (for instance, a "Double Latte" button). It also aims to provide an extensible framework to allow volume controls, multi-position switches or any other user interface widget to be defined in future Key Event Packages. A secondary goal of the proposal is that devices possessing a simple user interface (such as telephone dialpad) need only implement a smaller subset of the full proposal without losing any interoperability with Application Servers (c.f., section 10).

DTMF has been used in telephony networks as a means for users of network services to provide input and some control of those services. DTMF is also used to signal connection destinations in these networks. This dual role of DTMF does not present problems in circuit-switched networks. However, with telecommunications move to packet networks, separation of these roles is needed. Packet networks enable communications and related services to be deployed in a distributed architecture. An alternative to DTMF for "user input" is needed in order to better support the distributed application architecture possible in IP networks.

Although a standard mechanism exists for the transport of DTMF tones in IP networks, using it for user input and/or dynamic application control as is done in the PSTN does not carry over into the IP network very well. Receiving tones via RTP is problematic for the

detector/generator devices in scenarios where multiple network entities are interested in the events. Replicating media to multiple destinations is not a common feature of endpoints. In addition, the SDP session description to set up media replication is more complicated than that required to set up typical multi-media communications sessions.

The mechanism defined here can be used as an alternative to DTMF detection when the detection of these tones is to gather input from a user or provide some user application control independent of any end-to-end media path. The relationship of the mechanism described in this specification to the transport of telephony tones via RTP defined in RFC 2833 [4] is further described in Section 2.1.

It is intended that the event notification mechanism defined in this document might be used in scenarios that have the following attributes or requirements:

- The amount of data to be conveyed is very small compared to the audio, video, text, etc. data in a session.
- Dynamic user control of applications is required.
- Collecting device input is needed when the device is not engaged in a session.
- Security concerns exist due to multiplication attack possible with DTMF forking.
- End-to-end security/privacy between caller and destination system is required.
- Multiple Application Servers may be involved along the end-to-end call signaling path.
- Authentication and/or Privacy of key sequences between an Application Server and an endpoint is desired.
- Reliable delivery of key sequences is required even when short periods of network connectivity problems occur.
- Support for a device with a user interface consisting of more than a simple keypad is required.

A network entity subscribing to key events at another network entity may be interested in the events as they relate to an established communications session or as they relate to the device itself. Many network-based communications services require the user to identify themselves (for instance, by the user providing a PIN). Pre-paid and post-paid calling card services are a good example where a call-associated event subscription is useful. Key event subscriptions can also be useful outside of communications sessions. For example, an endpoint (SIP phone) may have a dedicated key used to asynchronously invoke some function on a remote network entity. In this case, the remote network entity subscribes to key events at the SIP phone in a non-dialog specific manner. When a key event of interest occurs, the remote endpoint can act on the event. RFC 2833

does not support notification of key events outside of the context of an established session.

2.1. Relation To Telephony Tones And Events

RFC 2833 defines an RTP payload format for telephony tones and signals (including DTMF digits). It allows the transport of such events in the context of an established media session (in SIP, a session must be established with media description in order for endpoints to exchange RFC 2833 events or tones). This mechanism has several advantages: it is independent from any signaling protocol and can be used with SIP, MEGACO, MGCP and other protocols, it eliminates tone distortion, and it provides a reliable means for telephony endpoints to exchange telephony tones within RTP media flows.

The purpose of the current document is to provide a general mechanism for SIP entities to request and report the notifications of key events (including keys in keypads or any kind of keyboards). This mechanism is specific to the SIP protocol and is based on SIP events (although the actual Key-Event Message Format is defined in a protocol independent manner). It offers several advantages:

- SIP application servers not involved in the media path can now request and receive key state changes.
- SIP entities supporting SUBSCRIBE/NOTIFY now have a generic mechanism to exchange key states.
- It allows multiple SIP application servers to receive independent key event notification with each application server only receiving notifications for the key-events that it is interested in.

In conclusion, this mechanism can be used in conjunction with RFC 2833 or it can also serve as an alternative to RFC 2833 in the cases where:

- Key states beyond telephony tones and signals are desired.
- A means for simple SIP telephony applications such as software phones or applications, not necessarily able to generate telephony tones, is needed to interact with applications.
- IP telephony gateways wish to conserve DSP resources.
- Supporting RFC 2833 for the sole purpose of indicating key presses, or supporting the duplication of RFC 2833 RTP payload events to the signaling planes, is not desirable.

2.2. Example Usage Scenarios

As discussed in the previous section, the scenarios described in this section do not preclude the presence of tone data associated with a telephone keypad. Specifically, RFC 2833 may still be employed as an end-to-end media-layer transport mechanism for DTMF tones, that is, as an end-to-end transport mechanism between the caller and destination User Agent.

2.2.1. Example 1: Application Server acting as B2BUA/3pcc

In this example the Application Server (AS) will generate the INVITE and BYE requests to perform call control. Due to the nature of this model, the AS can always inherently ensure that it remains on the call signaling path for the duration of the session. The general B2BUA network model is shown in Figure 1.

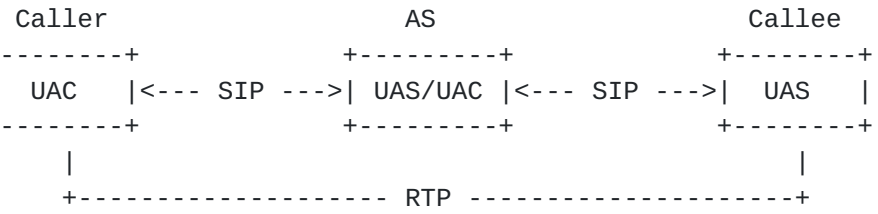


Figure 1. Application Server (B2BUA/3pcc) Model

The above figure depicts a caller establishing a session with the AS, after which the AS establishes a session with the callee. The AS provides one endpointÆs session description to the other endpoint. This results in the media being exchanged between the two endpoints and the SIP signaling occurring between the AS and each endpoint.

In this scenario, it is desired for the caller and callee to be able to invoke application services & features by pressing keys on the terminal. The action required by the AS is simple: it sends a SUBSCRIBE to the callerÆs and/or calleeÆs User-Agent as described in section 16.9. The subscription request should be dialog specific. Now the Application Server will receive all endpoint Key Events for the duration of the subscription.

2.2.2. Example 2: Application Server Acting As A Proxy.

In this scenario, the Application Server normally functions as a SIP proxy. The application is co-located with the SIP proxy and may also function as a SIP UA if desired. The network model is shown in Figure 2 below.

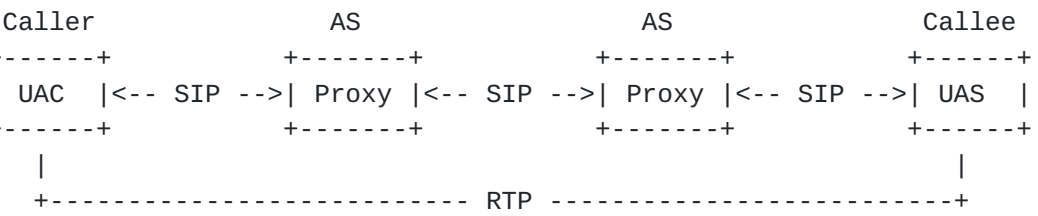


Figure 2. Application Server (Proxy) Model

Any proxy that wants to stay in the signaling path inserts a Record-Route header into the session establishment request. The Record-Route mechanism will ensure the Proxy/AS sees all SIP requests and responses for the dialog established between the caller and callee.

Applications deployed using this model do not manage user sessions as a user agent but rather exert call-control over the INVITE dialog and/or conversation space [5] using, for instance, the call control

primitives and framework defined in [5] or using some other external mechanism.

In this usage example the Application Server desires to take action after a certain sequence of key events occurs during a session. So after possibly record-routing the callerÆs initial INVITE request and forwarding it to the destination system, the AS will send a dialog specific subscription request to the desired User-Agents as described in Section 16.7 (and as for example 1). This allows any and all SIP proxies/AS in the signaling path to receive key-event notifications independently of any other SIP proxy/AS.

2.3. SIP Call-Control Interaction

The Key Event Package allows multiple network entities to subscribe to key events for the same SIP session on the same device. Due to this fact, it is possible for multiple Application Servers to take part in the same call and thus the interaction between the servers must be considered.

However, solving the problems of multiple AS interaction is beyond the scope of this document. This document simply defines a mechanism whereby Applications can monitor key events on a network device - it is does not specify the call-control mechanisms used by participating entities. Different call control models will have different interaction characteristics.

Part A -- Introducing a protocol independent Key-Event message format

The following section introduces a transport protocol independent message format for the requesting and transport of Key-Events. Part A of this draft is intentionally protocol neutral such that the resulting message format is adaptable to any reliable, bi-directional transport mechanism. Part B of this draft details the use of Part A within the context of a SIP Event Package [2] using the SIP SUBSCRIBE and NOTIFY methods (an example of a transport mechanism profile). If there is sufficient non-SIP related interest, Part A will be split out into separate internet draft.

The message format presented in Part A does not specify a complete protocol. However, the combination of the Key-Event Message Format and the transport mechanism profile (Part B) should specify a complete protocol.

When the Key-Event Messages are transported as MIME message bodies, the MIME content type for this message format is "application/key-event" and is officially defined in section 12.

3. Terminology

To avoid confusion and to maintain a protocol neutral terminology, the following terms will be used in this section of the document.

Requestor: the entity requesting Key-Event notification from a target entity (the Notifier).

Notifier: the entity to which the Requestor is requesting to be notified of Key-Events.

Transport Mechanism: The protocol mechanism used to reliably route and deliver messages between the Requestor and Notifier. It is the responsibility of the transport mechanism to inform the Requestor and Notifier when a Key-Event Session is terminated due to network or endpoint failure. The transport mechanism should also provide a mechanism whereby the Notifier (and optionally the Requestor) can indicate its desire to terminate an active Key-Event Session. The transport mechanism is not prescribed by the Key-Event Message Format.

Subscription: the state information held by the Notifier regarding the key event notifications successfully requested by a particular Requestor.

Key-Event Session: the time period when a subscription for key-events has been established (or in the process of being established).

4. Key-Event Messages

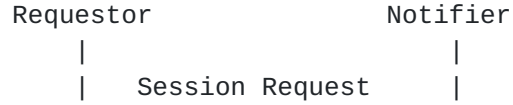
The Key-Event Message Format consists of the following three message types:

Key-Event Session Request: The Requestor sends a Session Request to the Notifier to request key-event notification and/or determine a NotifierÆs capabilities. In effect the Session Request creates a subscription to a set of key-events for which the Notifier will send notifications.

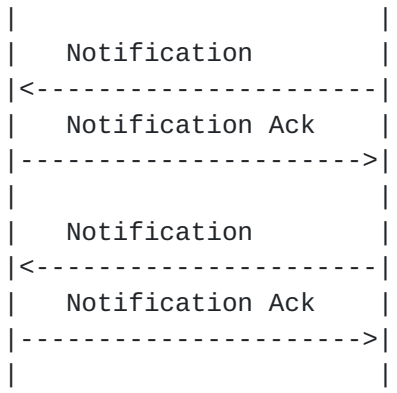
Key-Event Session Response: The Notifier, in response to a Session Request will send this message to indicate the full state of the subscription created by the Session Request and the current full state of all subscribed key-events.

Key-Event Notification: A Notifier normally sends a Notification whenever the state of any subscribed key-event changes. Each notification contains a state delta from the previous Key-Event Notification message.

A typical Key-Event Session message flow:




Session Response



A Key-Event Session can be renegotiated or terminated by the Requestor by sending another Session Request message. This Session Request message replaces all state established by the previous Session Request and will always solicit a Session Response. A Session Request with no requested events terminates a Key-Event Session.

5. Key Representations

The representation of keys in the key-event message format has been designed around a few central philosophies.

Firstly, all keys that represent the same information must map to the same key-event. For instance, a key representing "1" on a keyboard, keypad, telephone, or vending machine will all map to the same key event; likewise, multiple equivalent keys (e.g., left and right shift keys) will also map to the same key-event. The key-event package does however allow an optional positional indicator to indicate the origin of a particular key event (if multiple equivalent keys are present). The value can be ignored if desired. This arrangement avoids the situation where an application must observe a (possibly unknown) set of equivalent key events.

The positional indicator values currently defined are "alt" (alternate) & "keypad" as well as the assumed "pri" (primary) key position. When a keyboard has two or more equivalent keys, one is always defined to be the primary key and the others are defined to be the alternate/keypad positions. For Shift, Ctrl, Alt and similar equivalent keys, the bottom-left most key is defined to be the primary key. If a device has only one instance of a particular key it is always regarded as the primary key regardless of its physical position.

The second design philosophy is that the event package is independent of key-cap arrangements on keyboards. A keyboard is represented by two distinct sub-packages: one sub-package represents all characters that are accessible on the keyboard, the second sub-package represents all non-character based keys, for instance,

Shift, Caps Lock, Insert, Up, Alt, Pause. [For historical reasons Backspace, Delete, Enter/Carriage-Return & Escape have character representations.] This, for example, means that the representation of capital Q (i.e., 'Q') is independent of the Shift key event.

Lastly, all basic keys were designed to have two possible states: up or down. Key presses are signaled by notifying the requestor when a key transitions into a new state ("keyup" or "keydown" state events). Each state notification is accompanied by state specific information such as key depression start time. The key-event package states have been arranged such that a "keyup" state event includes both key release event time as well as duration of the depression.

This arrangement allows the "keydown" state event to be seamlessly delayed or omitted by the Notifier and/or ignored by the Requestor. The role of the "keydown" event is to provide rapid notification of key depression. For many applications, the "keydown" event notification is only useful for prolonged key depressions. The key-event package provides a useful mechanism whereby a Requestor application can delay "keydown" notifications to achieve a compromise between key response times and reduced bandwidth (1 or 2 notify messages per key depression and release). [c.f., Section 7.1 "dwndly" property.]

6. Key-Event Framework

The Key-Event Message Format does not in itself define key events but rather defines the framework for specifying "Key-Event Sub-Packages". Each Key-Event Sub-Package defines the key-events that make up the sub-package along with the states and properties that the constituent key-events may possess.

In section 7, this document defines Key-Event Sub-Packages for standard keyboards, telephones and user-defined buttons. Section 11 provides guidelines for future sup-package authors.

7. Key-Event Sub-Package Definitions

All Key-Event Sub-Package definitions consist of the following components:

Sub-Package Name: The name of the sub-package as used in messages.

Key-Event Values: This list defines the set of valid key-events of the sub-package. A key-event can have either a numeric value (e.g., 49) or a non-numeric value (e.g., shift).

Key-Event States: This component defines the set of states that key-events in the sub-package can have (e.g., keyup or keydown). The sub-package must also define a default state.

Key-Event Properties: This list defines any configurable or retrievable properties held by the key-events. Support for any

particular sub-package property is NOT REQUIRED unless explicitly stated in the sub-package description.

Enumerated Key-Event Sets: In order to simplify subscription, the sub-package can define a number of enumerated key-event sets. These enumerations are entirely equivalent to the set of key-events they represent (e.g., @dialpad = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, * and #} found on common telephones).

By design, all key-events in a sub-package share the same states and properties.

7.1. Shared Key-Event States and Properties

All currently defined sub-packages in this document share the following state and property definitions.

These definitions have been grouped together for clarity however the definitions still belong to the individual Key-Event Sub-Packages. Future sub-packages may define additional states and/or properties; future sub-packages are also not forced to use existing key-event states and properties if they do not make sense within a new sub-package (e.g., for a slider/dimmer control). (c.f., Section 11 for further guidelines.)

Key-Event States:

- "keydown"
  State Parameters:
    * Start Time of Key Depression: Milliseconds offset from the event-time NTP value specified in the Time-header. Negative values are allowed.

- "keyup"
  State Parameters:
    * Time of Key Release: Milliseconds offset from the event-time NTP value specified in the Time-header. Negative values are allowed.
    * Duration of Key Depression: Millisecond duration of key depression.

Key-Event Properties:

- "exists"
  Request Parameters: none
  Response Result: positional-indicator *("," positional-indicator)
  [List of non-primary key instances.]
  Default value: N/A
  Support: RECOMMENDED
  Description: Determines which non-primary key-events exist.
  e.g., "p: key[49].exists()" may yield a result of
  "p: key[49].exists(keypad)" which indicate that two '1' keys exists: a primary key instance and a keypad key instance.

Note: This property only produces a result for key-events possessing non-primary key positions.

- "name"
  Request Parameters: none
  Response Result: positional-indicator "=" quoted-string
  *("," positional-indicator "=" quoted-string)
  [Lists of names for all key instances.]
  Default value: N/A
  Support: RECOMMENDED
  Description:
  The property when requested (e.g., "p: keybd[49].name()") returns the printable name of the key-event in the Session Response result (e.g., "key[49].name(pri="1", keypad="Keypad 1")").
  The description should be sufficiently detailed such that a user would be able to identify the key from the description. This property can also very useful when the name of the key is configured locally, for instance, a Session Request for "phone[line:*].name()" could conceivably yield a result of "phone[line:1].name(pri="Reception")".

- "dwndly": (DownDelay)
  Request Parameters: 1*DIGIT (milliseconds delay)
  Response Result: 1*DIGIT (milliseconds delay)
  Default value: 0
  Support: OPTIONAL
  Description: Milliseconds delay that Notifier should/will wait before generating a "keydown" state event after a key is depressed. If the key is raised before this duration elapses then only the "keyup" state event will be generated. This allows a tradeoff between key responsiveness (for longer keypresses) and the number of Key-Event Notification messages per keypress. The Session Response result contains a confirmation of the new value.
  This value (and any change thereof) is local to the Key-Event Session in which it is requested.

7.2. Key Sub-package

Sub-Package Name: "key"

Key-Event Values: Decimal Unicode values [7].

Key-Event States: keydown, keyup (default)

Key-Event Properties: name, exists, dwndly

Enumerated Key Event Sets:
- "@dialpad" = "35,42,48-57"
  This set represents a standard telephone dialpad: 0-9, *, #.
- "@uslatin" = "9-10,32-126"

This set represents the printable and white space character key set found on a typical US keyboard (which is a subset of most other keyboards).

- "@uslatin2" = "8-10,27,32-127"

This set is similar to @uslatin but also includes the non-printable backspace, delete & escape keyboard characters (see notes below).

Notes:

For mainly historical reasons, the following keys belong to the key sub-package rather than the keybd sub-package:

- Backspace: 8
- Tab: 9
- Enter: (treated as unix '\n') 10
- Escape: 27
- Delete: 127

7.3. Keyboard Sub-Package

Sub-Package Name: "keybd"

Key-Event Values:

"shift": name="Shift" (or "Left/Right Shift" if applicable)
"caps": name= "Caps Lock"
"ctrl": name="Control"
"alt": name="Alt"
"home": name="Home"
"end" : name="End"
"insert": name="Insert"
"pgup": name="Page Up"
"pgdn": name="Page Down"
"up": name="Up"
"dn": name="Down"
"right": name="Right"
"left": name="Left"
"pause": name="Pause"
"numlk": name="Num Lock"
"scrlk": name="Scroll Lock"
"prtsc": name="Print Screen"
"break": name="Break"
+ "func:*": name="Function Key x"
+ "os:*": name=<operating system defined>
+ "vendor:*": name=<vendor defined>

+ Key-events of the form "keyname:*" allow for multiple numbered keys to exist. These key-events must be always be used in this numbered key form (e.g., "func:3").

Notes:

The names provided for the Key-Event Values should be only regarded as examples. The actual Key Event Name should be sufficient to locate the individual key on the device (including the case when there are multiple key instances).

Key-Event States: keydown, keyup (default)

Key-Event Properties: name, exists, dwndly

Enumerated Key-Event Sets:
- "@basic" = "shift, caps, ctrl, alt, home, end, insert, pgdn, pgup, up, dn, right, left"
  (basic keyboard)
- "@std" = "shift, caps, ctrl, alt, home, end, insert, pgdn, pgup, up, dn, right, left, pause, numlk, scrlk, prtsc, break"
  (standard keyboard)

7.4. Telephone Sub-Package

Sub-Package Name: "phone"

Key-Event Values:
"hook": name = "Hookswitch"
Notes: The state of this button is actually the opposite of the hook switch, that is, the "keyup" state represents the on-hook condition. Normally this key-event would only be available in non-dialog specific subscriptions.
"hold": name = "Hold"
"conf": name = "Conference"
"flash": name = "Flash"
"redial": name = "Redial"
"transfer": name = "Transfer"
+ "line:*": name = "Line x" or locally defined
+ "speed:*": name = "Speed Dial x" or locally defined
+ "vendor:*": name = <vendor defined>

+ Key-events of the form "keyname:*" allow for multiple numbered keys to exist. These key-events must be always be used in this numbered key form.

Key-Event States: keydown, keyup (default)

Key-Event Properties: name, exists, dwndly

Enumerated Key-Event Sets: None.

7.5. User Package

This sub-package intentionally does not define any key-event values. This package is designed as a sub-package available for any device or user specific key-events and properties.

Key-Event Values: None. (see note)

Key-Event States: keydown, keyup (default)

Key-Event Properties: name, exists, dwndly

Enumerated Key-Event Sets: None. (see note)

8. Key-Event Message Body Format

A Key-Event message body consists of a set of UTF8 header lines. The header lines resemble HTTP header formats however for efficiency all header, sub-package, property, state and key-event names are case-sensitive and multiple headers of the same type are not permitted. Parsers MUST be tolerant of spaces between header tokens.

```
key-event-message = time-header CRLF
                    [ instance-header CRLF ]
                    [ event-header CRLF ]
                    [ properties-header CRLF ]
                    [ state-header CRLF ]
                    [ future-extension-header CRLF ]
```

The individual key-event headers may appear in Key-Event messages as detailed in the following table:

Key-Event Header	Request	Response	Notification
Time-Header (t:)	m	m	m
Instance-Header (i:)	o	-	-
Event-Header (e:)	m	m	-
Properties-Header (p:)	o	o	-
State-Header (s:)	-	o	m

m = mandatory; o = optional; otherwise, the header is not allowed.

8.1. Time-Header Format

The time-header indicates the NTP time [6] that the event message was first issued and a message sequence number.

The exact starting/ending time of an event often requires greater granularity than is provided by NTP values (in seconds) and so key-event state transitions also include a millisecond offset that is added to this NTP time value. It must also be remembered that the time a key-event message is generated may not always be the same as the time of the actual key-event event (for instance, initial state indications).

The sequence number allows the key-event message bodies to be sequenced independently of the transport protocol.

```
time-header = "t:" event-time sequence-number
event-time = 1*DIGIT
sequence-number = 1*DIGIT
```

The event-time is set equal to the NTP time value when the message

body was first issued. The sequence-number is a 32-bit non-zero integer that is incremented each time a new message is generated

within each key-event session. The sequence number spaces for the Requestor and Notifier are independent.

Examples:

t: 36539655 1

8.2. Instance-Header Format

This optional header allows for a protocol specific identifier to be placed in a Session Request message to identify the target endpoint instance (and association parameters) when the transport mechanism cannot provide such functionality.

```
instance-header = "i:" *( ";" | "/" | "?" | ":" | "@" | "&" | "="
                  | "+" | "$" | SP | TAB | unreserved | escaped )
```

The headerÆs format and interpretation must be specified in the transport mechanism profile if used.

8.3. Event-Header Format

The event-header allows the Requestor to request a set of key-events and the Notifier to confirm/indicate the set of key-events subscribed/supported.

The event-header in a Session Request is used to request a subscription to a set of key events. The event header in a Session Response is used to indicate the actual set of subscribed key events.

```
event-header = "e:" [ ( event-set ( "," event-set ) ) | "*" ]
event-set = sub-package-name [ key-event-list ]
sub-package-name = token
key-event-list = "[" key-event-set *( "," key-event-set ) "]"
key-event-set = key-event-identifier | key-event-range |
                enumerated-key-event-set
key-event-identifier = key-event-name | key-event-value |
                       key-event-numid | key-event-numidwild
key-event-name = token
key-event-value = 1*DIGIT
key-event-numid = token ":" 1*DIGIT
key-event-numidwild = token ":" "*"
key-event-range = ( key-event-value "-" key-event-value ) |
                  ( key-event-numid "-" key-event-numid )
enumerated-key-event-set = "@" token
```

If the key-event-list is not present in a Session Request message (e.g., "e: key") then it is assumed that the Requestor wants to be notified of all available key-events in the sub-package. The key-

event-list MUST always be present in Session Response messages and indicates all key-events that are subscribed (e.g., e: key[48-57]).

Likewise, the "*" wildcard can only appear in Session Request messages. Wildcards must be expanded in Session Response messages as described in Section 9.2.

Key-event-ranges with numbered keys (e.g, a "Line 3" telephone button) must have the same key-event type on both sides of the range (e.g., "phone[line:1-line:10]"). Each numbered key is treated as an independent key-event.

An empty event-header ("e:") in a Session Request or Response message indicates the termination of the Key-Event Session. The event-header is the only key-event header which may be present but empty.

Example Session Request message headers:

e: key[@dialpad, 65-68], keybd

In this example, the Requestor is requesting event notification for keys: 0-9, *, #, A, B, C, D from the key sub-package and all supported keybd sub-package events.

e: *

In this example the Requestor is requesting a subscription to all key-events supported by the Notifier.

e: phone[line:*]

In this example the Requestor is requesting all available "line" buttons within the phone sub-package.

Example Session Response headers for the above Session Request message headers:

e: key[@dialpad]

In this example the Notifier is indicating that subscription only occurred for the key-events 0-9,*,#.

e: key[48-57,8,10], keybd[insert,numlk]

In this example, the Notifier is indicating that subscriptions have been created for the key-events 0-9, Backspace, Enter, Insert & NumLock.

e: phone[line:1-line:5]

In this example, the Notifier is indicating that subscriptions have been created for the "line" buttons 1 through 5.

8.4. Properties-Header Format

The properties-header in a Session Request message is used to set or retrieve the properties of subscribed key-events. The properties-header in a Session Response message indicates the value of each of the listed properties.

```
properties-header = "p:" properties-set *( "," properties-set )
properties-set = ( event-set | "*" ) "." property-name
                 "(" [ parameter-list ] ")"
property-name = token
parameter-list = parameter *( "," parameter )
parameter = *( ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"
         | "$" | SP | TAB | unreserved | escaped | quoted-string )
```

As with the event-header, "*" wildcards and empty key-event-listÆs can only appear in Session Request messages. Wildcards and key-event-listÆs must be expanded in Session Response messages as described in Section 7.2.

NOTE: The set of key-events that each property operates on is always equal to the intersection of the set of subscribed key-events and the set of key-events requested in the properties-header.

Examples for Session Request message headers:

p: key.dwndly(500)

In this example the Requestor is requesting that a value of 500 is assigned to the "dwndly" property for all subscribed "key" sub-package key-events.

p: *.exists()

In this example the Requestor is requesting to know the existence of all non-primary key instances for all subscribed key-events.

p: keybd[ctrl].name()

The Requestor is requesting the name of all ctrl key instances.

Example Session Response message headers for above Session Request message examples:

p: key[@dialpad].dwndly(500)

In response to the above dwndly set request, the Notifier is confirming the new value for all subscribed key-events in the key sub-package.

p: key[47-58, 10].exists(keypad)

The Notifier is indicating that in addition to the primary instances of all subscribed key-events, additional key instances for 0-9 & Enter exist on a keypad.

```
p: keybd[ctrl].name(pri="Left Control", alt="Right Control")
```

In this example, the name property indicates the names for the various control key instances possessed by the device.

8.5. State-Header Format

The state-header allows the Notifier to indicate the initial key-event states and subsequent key-event state changes. The header only appears in Session Response and Notification messages. In a Session Response message the state header gives key-event state information for all subscribed key-events in non-default states. In a Notification message the state header only indicates a change in key-event state.

```
state-header = "s:" state-indication *( "," state-indication )
state-indication = event-descriptor "." state-name
                   "(" [ parameter-list ] ")"
event-descriptor = sub-package-name "[" key-event-descriptor "]"
key-event-descriptor = ( key-event-name | key-event-value |
                       key-event-numid )
                       [ "(" positional-indicator ")" ]
positional-indicator = "pri" | "alt" | "keypad" | token
state-name = token
```

An empty positional-indicator infers the request/result applies to the primary key of this key-event type.

Examples:

```
s: key[49].keydown(-1532)
```

This example is a keydown state indication for the primary '1' key (Unicode [7] value 49). It indicates that the button was first depressed 1532 milliseconds before the NTP value in the time-header.

```
s: key[49(keypad)].keyup(126,1206)
```

This example is a keyup state indication for the keypad '1' key. It also indicates that the button was released 126 milliseconds after the NTP value in the time-header and was depressed for 1206 milliseconds.

9. Key-Event Message Generation

9.1. Generating a Session Request message

The Session Request message is generated to establish, renegotiate

or terminate a Key-Event Session. The values placed in the messageÆs time-, instance-, event- and properties- headers are described in the relevant sections of section 8.

9.2. Generating a Session Response message

A Session Response message is generated in response to a Session Request. Generating the Initial NOTIFY request is the most complicated part of this specification, however, the headers were constructed so that the event-, property- and state- headers have a similar syntax and semantics.

The procedure for generating the Response is decomposed into the procedures for generating each of the possible Session Response message headers.

9.2.1. Event Header Processing

This section describes how the subscribed key-events set is generated and how the Session Response event-header is derived from this set. The subscribed key-events set also forms part of the subscription state information and is used in generating the other headers. The set is formed from the event header in the Session Request message.

To summarize, the subscribed key-events set is formed by expanding all wildcards in the requested key-events set and intersecting the resulting key-event set with the supported key-event set.

Procedure:

An event header of "*" results in a subscribed key-event set equal to the supported key-event set. If the event header does not contain a "*" then the following procedure is performed for each element on the event header line:

If the key-event-list for the sub-package is empty (e.g., "e: user") then all supported key-events in the specified sub-package are added to the subscribed key-events set. Otherwise, the following procedure is performed for each element in the key-event-list:

For enumerated key-event sets (e.g., "@dialpad"), if all key-events in the set are supported, then the enumerated key-event set is added to the subscribed key-events set. Otherwise the enumerated key-event set is simply replaced by its equivalent key-event list for individual evaluation.

All key-event identifiers and ranges are checked against the supported key-event set and supported key-events are added to the subscribed key-event set.

The resulting set is the subscribed key-event set.

The event header in the Session Response message is generated from the subscribed key-events set by grouping together all key-events in the same sub-package. This is done to keep header length down, for

example, "key[47-58,10,35]" is preferred over "key[47], key[48], key[49], ...".

If the subscribed key-event set is empty then an empty event-header is generated ("e:") and the subscription is terminated after the message is delivered.

9.2.2. Properties Header Processing

The properties header in the Session Request message is evaluated to form the properties header used in the Session Response message as follows:

Each element in the Session Request properties header is checked. If the property is not supported for any sub-package then no action is taken for this element. Otherwise, the input key-event set is calculated (the key-event set on which the property will operate). This set is generated in a similar manner to the subscribed key-event set described in the previous section except that the subscribed key-event set is used instead of the supported key-event set (properties only operate on subscribed key-events) and the requested key-event set is equal to the event-set of the current properties header element. The resulting input key-event set is further reduced by removing all key-events for which the property is not supported. At this point, the property is evaluated for each element in the input key-event set. For each evaluation, if the property generated a result then the result is added to the output set, otherwise no action is taken.

Like the event header, the properties header is generated from the output set by grouping together all key-events with the same property result (and in the same sub-package). For example, "key[47-58].exists(keypad)" is preferred over "key[47].exists(keypad), key[48].exists(keypad), key[49].exists(keypad), ...".

If the output set is empty then the properties header is not added to the message.

9.2.3. State Header Processing

A state notification is generated for each key-event in the subscribed key-event set for which the current state is not equal to the default state. This set of state notifications forms the Session Response state-header.

If all subscribed key-events are in the default state then a state-header is not added to the message.

9.3. Generating Key-Event Notification messages

Notification messages generated in response to key state changes only possess two headers: the time-header and the state-header.

The information in the state-header is a delta indication of the key-event state. In other words, each Notification message only includes the state information of key-events that have changed since the last notification.

10. Minimal Implementation for Key-Event Notifiers

Firstly, even a minimal implementation MUST address the security considerations prescribed in each transport profile.

When a device does not have any user-defined keys, locally-configured names, multiple key instances or numbered buttons (e.g., a simple DTMF telephone dialpad) then the implementation can be made quite simple. In these cases the "exists" and "name" properties can be deemed OPTIONAL. If remote configuration of "dwndly" is also not required then the properties header line MAY be ignored completely.

This allows simple devices to scale down to a simple implementation without affecting the functionality available to more complicated user interface devices.

11. Guidelines to Future Sub-Package Authors

Future sub-packages must not duplicate key-events contained in existing sub-packages.

If the relevant sub-package already exists then an extension for the existing sub-package should be proposed rather than a new sub-package.

Once published as a standard, new key-event states or enumerated key-event sets SHOULD NOT be added to an existing sub-package. Only new key-event values and OPTIONAL key-event properties may be added to existing sub-packages.

New sub-packages should reuse the states and properties of the existing sub-packages where possible.

12. MIME type information for the Key-Event Message Format

The MIME definition for the key state message bodies defined in this document follows.

Media type name: application
Media subtype name: key-event
Required parameters: none
Optional parameters: none
Encoding scheme: text

The media subtype is used to indicate the message content as defined in this document. A typical header would look like the following.

Content-Type: application/key-event

Part B -- The use of SIP SUBSCRIBE & NOTIFY for Key-Event Transport

This section describes the transport of key state data defined in Part A using the SIP Event Notification framework.

13. Subscription to Key Events

Subscriptions to key events are established, maintained, and terminated as described in the SIP-Specific Event Notification framework specification [2]. Key event subscriptions are indicated using "key-event" for the event-type in the Event header of SUBSCRIBE and NOTIFY requests. Details of the events being subscribed to and being reported are specified in message body of the Key-Event Session request. (See sections 7 and 8 for normative descriptions of Key-Event Sub-Packages and their message bodies.)

As specified later, SUBSCRIBE requests contain Key-Event Session Request message bodies that indicate the events being subscribed to. Once accepted, an "event notification dialog" is established and event state is established in the Notifier according to the contents of the SUBSCRIBE message body. It is possible to modify the set of events being subscribed to by sending another SUBSCRIBE request, with a key events message body, for a previously established event notification dialog. In this case, the new message body should replace the state established by a previous SUBSCRIBE.

This document describes two types of key-event subscriptions. The first is a subscription associated with a SIP dialog or conversation space. That is, the subscription is only valid for key-events directly associated with the specified session and the subscription ends when the SIP dialog or conversation-space ceases to exist. The term "dialog specific subscription" will be used to describe this type of subscription. The second type of key-event subscription described here is a non-dialog associated subscription. Where the subscribing device is interested in key events regardless of what the device is doing. The term "non-dialog specific subscription" will be used to describe this second type of key event subscription. Devices supporting the mechanisms defined in this document MUST support dialog specific subscriptions. Support for non-dialog specific subscriptions is OPTIONAL.

13.1. Non-Dialog Specific Subscriptions

Key-Event subscriptions that apply to a device, regardless of any on-going or active SIP dialogs, are established by SUBSCRIBE requests that do not include an instance-header in the Session Request message body. Notifiers that support non-dialog specific

subscriptions MUST support multiple concurrent subscriptions from the same or different subscribers.

13.2. Dialog Specific Subscriptions

A dialog specific subscription is requested by the subscriber by including an instance-header in the Session Request message body of the SUBSCRIBE request. This header identifies the particulars of the dialog specific subscription; the format of the inserted header is described in section 14.

There are two modes of dialog specific subscriptions: dialog attached mode and conversation-space attached mode. The primary difference between the two modes is the conditions under which the subscription terminates.

13.2.1. Dialog Attached Mode

In this mode, the subscription receives key-events from the user endpoint associated with the specified dialog whenever the user is *actively* communicating through that user endpoint. [An example of a user endpoint in this context could be a single virtual "line" on a multi-line SIP phone.]

The subscription is terminated when the dialog is terminated. The RECOMMENDED subscription expiry timeout is on the order of a day. When a subscription is terminated due to dialog termination, the Notifier MAY send a NOTIFY with "Subscription-State: terminated;reason=noresource" header however this is deemed unnecessary in most cases.

13.2.2. Conversation Space Attached Mode

Just as for dialog attached subscriptions, the subscription receives key-events from the user endpoint associated with the specified dialog (at the time of subscription) whenever the user is *actively* communicating through that user endpoint.

However for conversation space attached dialogs, the subscription is terminated when the conversation space of the user endpoint is reduced to a conversation space containing only the user endpoint (or an empty conversation space). For instance, in this mode the subscription would *not* be terminated when an INVITE with a Replaces header was received from the network but would be terminated when all other participants leave a distributed conference. It is RECOMMENDED that the subscription expiry timeout have a value ranging from minutes up to an hour. The Notifier MUST send a NOTIFY with "Subscription-State: terminated; reason=noresource" header when the subscription terminates.

13.2.3. Multi-dialog Capable Endpoints

Dialog specific subscriptions can present a challenge to user

devices that support multiple simultaneous dialogs but only have a single user interface (keypad, speaker, microphone) that is shared between all session (e.g., a multi-line SIP phone). Most devices of

this type have the concept of a single "active" session among the established sessions. That is, only one session has the use of the shared user interface resources at any point in time. This characteristic should be applied to key event subscriptions, in other words, notifications will only be sent to subscribers (if any) of the dialog with which the keypad is assigned to at the time of the event detection. This behavior is consistent with common circuit-based multi-line telephones.

This should not be confused with devices which terminate multiple dialogs but where each dialog is associated with a different user, for instance, a SIP-PSTN gateway.

13.2.4. "Early" Dialog Subscriptions

In order for a subscriber to reliably receive all Callee generated key-events from the time a session is fully established/answered, a Callee must be able to receive and setup a dialog specific subscription prior to the associated dialog being fully established. This is accomplished by the caller/AS sending a SUBSCRIBE request after an early dialog has been established (by a provisional INVITE response) but before any final response has been sent by the callee. As with any SUBSCRIBE request received, recipients should process the request according to its policies, however early dialog subscriptions do not produce Key-Event information until the associated dialog is established/answered. It should be remembered that the SIP Events draft requires that a NOTIFY request is generated immediately, however, the NOTIFY will simply not contain Key-Event message bodies until the subscriptionÆs associated device or dialog becomes active.

A Caller can also receive a dialog specific subscription for a dialog before it has been fully established. The same issues with authentication and authorization apply as in the previous case, however, the notification of caller key presses before the dialog is fully established may benefit certain applications (for instance, pre-answer or pre-alerting caller pin-code authorization). Thus, when the instance-header does not specify a remote tag the early subscription should become active immediately. If the subscriber does specify a remote tag in the instance-header, then the subscription is only valid while the caller has an end-to-end association with the specified callee (for instance while early media from the specified callee is being played to the caller). In this case, the subscription is terminated if and when the session is eventually established with a different callee.

13.2.5. Multiple Subscriptions

A User-Agent/Notifier MUST support multiple concurrent subscriptions

for the same dialog from both the same subscriber and from different subscribers.

14. Instance-Header format for SIP Dialog Specific Subscriptions

The purposed of an instance-header in the key-event message body of a SUBSCRIBE request is to identify a SIP dialog and specify the subscription attachment mode; in a SIP SUBSCRIBE, the headerÆs presence indicates that the subscription is dialog specific.

If a subscriber places an instance-header in the initial SUBSCRIBE request then it MUST also be placed unchanged in all subsequent SUBSCRIBE requests for the SIP dialog specified initially. However, a Notifier MUST ignore the instance-header in all but the initial SUBSCRIBE request.

> This allows a Notifier to restart but ensures that the subscription does not change over time. Notifiers can ignore the instance-header in SUBSCRIBE requests that refresh subscriptions since SUBSCRIBE established dialogs are identified by the Call-ID, Local-Tag, and Remote-Tag present in the initial SUBSCRIBE request.

```
instance-header = "i:" instance-component *("," instance-component )
instance-component = "call-id" "=" Call-ID
                   | "local-tag" "=" UUID
                   | "remote-tag" "=" UUID
                   | "attach" = ( "dialog" | "cspace" | token )
                   | future-extension
```

No minimum set of components has been specified for the instance-header - it is the responsibility of the generator to ensure that the instance-header uniquely identifies a dialog on the target. To ensure uniqueness it is RECOMMENDED that the instance-header contain the call-id, local-tag and remote-tag components.

If none of the "call-id", "local-tag", "remote-tag" components are present then it is assumed that the dialog specific subscription is associated with the subscriptionÆs own dialog "call-id", "local-tag" & "remote-tag" values. This does not preclude the use of the attach component.

The "attach" component indicates whether the subscription is associated to the dialog itself or to the conversation space to which the dialog currently belongs (as described in section 13.2). The assumed value for the "attach" component (when not present) is "attach=dialog".

15. SIP specific Key-Event error handling

15.1. Handling of unacceptable instance-header values.

When a SUBSCRIBE request is received containing an instance-header

referring to a non-existent or unacceptable dialog then the Notifier SHOULD generate a NOTIFY request with a "Subscription-State: terminated;reason=noresource" SIP header.

15.2. Handling of unknown key-event-set enumerations.

When a Notifier receives a Key-Event Session Request with an unknown key-event-set enumeration in a supported key-event sub-package (e.g., "e: key[@polish]") then the Notifier MUST generate a SIP Warning header in the Initial NOTIFY request. Unknown key-event-set enumerations in the "user" sub-package MAY be silently ignored.

The "warn-text" of the SIP Warning header SHOULD include the unknown enumeration, for instance:

Warning: 390 host53.company.com "Unknown key-event-set enumeration: key[@polish]"

The subscription should proceed as normal with the unknown key-event-set enumerations ignored.

A warn-code of 390 is chosen as the information to be conveyed falls in the miscellaneous category (as defined in [3]) and to allow automated actions to occur. For example, a subscriber can modify the subscription and use Key-Event values instead of using an enumerated Key-Event set.

16. Event Package Definition

This section contains the formal definition of the Key-events SIP Event Package described in this document. This information is provided to conform to the requirements for SIP Event Packages as outlined in [2]. As part of these requirements this section details how Key-Event message bodies are generated by the subscriber and Notifier entities.

16.1. Event Package Name

The event package token name for the Key Events package is "key-event". The token name is used in the Event and Allow-Event headers defined in [2].

16.2. Event Package Parameters

There are no parameters used in the Event header for the key-events package/sub-packages.

16.3. SUBSCRIBE Bodies

SUBSCRIBE requests for Key-Event subscriptions MUST contain a Key-Event message body if the SIP Expires header value is non-zero and MAY contain one if the SIP Expires header value is zero. SUBSCRIBE responses do not contain a message body. The Content-Type for the

Key-Event message body is "application/key-event". The grammatical specifications for the message body are defined in section 8.

16.4. Subscription Duration

Dialog specific subscriptions implicitly end when the associated dialog or conversation-space ends. Please refer to section 13.2.1 & 13.2.2 for recommended dialog specific subscription expiry timeout values. It is recommended that the subscription expiry timeout for non-dialog specific subscriptions be from a few minutes to an hour.

As detailed in [2] a SUBSCRIBE with an "Expires: 0" header indicates the termination of a subscription. However, if the accepted SUBSCRIBE message also contains a Session Request message body, then the SUBSCRIBE message body should be processed and the normal NOTIFY Session Response message body is generated (ignoring any persistent subscription side effects). This mechanism can be used to determine the sub-packages & key-events supported as well as the current property values without creating an ongoing subscription.

If the terminating SUBSCRIBE does not contain a Key-Event message body then a NOTIFY request MUST still be sent (with no message body) as described in [2].

16.5. NOTIFY Bodies

NOTIFY requests MUST contain a Key-Event message body if the associated device, dialog or conversation space of the subscription is in an active state. Conversely, the lack of a message body in a NOTIFY request with a non-zero Subscription-Expires header value indicates that the Key-Event subscription state is pending (for example, awaiting authorization or for the associated dialog to become active). NOTIFY responses do not contain a message body. The Content-Type for the Key-Event message body is "application/key-event". The grammatical specifications for the message body are defined in section 8.

16.6. Notifier Processing of SUBSCRIBE Requests

The Notifier takes the following basic steps:

1. Before processing a received SUBSCRIBE request a state-agent MUST check that the subscriber is authorized to perform the requested subscription (c.f., Section 18 and [2]).

2. If the SUBSCRIBE includes an instance-header in the key-event message body then the Notifier MUST check that the dialog exists and if not, take action as described in section 15.

3. The SUBSCRIBE message body is parsed for consistency and a SUBSCRIBE response is generated as described in [2].

4. The state-agent will then clear existing Key-Event subscription

state for this subscription and proceed to generate a Key-Event Session Response message body in a NOTIFY request as described in Section 16.7.1.

5. If the value of the SUBSCRIBEÆs Expires header is non-zero then the new subscription state information is stored.

16.7. Notifier Generation of NOTIFY Requests

This section describes both the generation of NOTIFY requests when the subscription is established or modified by the Notifier, and when reporting state changes in the subscribed resource.

16.7.1. Generating Initial NOTIFY Requests

The Initial NOTIFY request is generated when the subscription state initially transitions to "active" or a subsequent SUBSCRIBE message is received by the Notifier. The initial NOTIFY request may only contain a Session Response message body which is generated in response to the accepted SUBSCRIBE request (although it need not be generated immediately). The Initial NOTIFY request always reports complete state information and can be differentiated from a NOTIFY reporting a change in state by the presence of the event-header in the Key-Event message body.

If the SUBSCRIBE request did not contain a Key-Event message body (which infers that the request also has an "Expires: 0" header) then the resulting NOTIFY request will also not have a Key-Event message body. If the SUBSCRIBE request does contain a Session Request message body but has a "Expires: 0" SIP header then an Initial NOTIFY Session Response request is still generated however the subscription is not kept after the Initial NOTIFY request is sent.

The Key-Event Session Request message body in the Initial NOTIFY request contains the following information:
- A confirmation of the subscribed key-events (event-header).
- The result of all key-event property evaluations (properties-header).
- The initial state of subscribed key-events that have a non-default state (state-header).

16.7.2. Generating Subsequent NOTIFY Requests

NOTIFY requests that are not generated in response to an accepted SUBSCRIBE request MAY include a Key-Event Notification message body to indicate a change in key-event state.

Once a NOTIFY request with an application/key-event message body has been sent, further "key-event" NOTIFY messages can be delayed until a final response is received for the NOTIFY request. Thus, the key-event state-header MAY contain multiple key-state changes.

16.8. Subscriber Processing of NOTIFY Requests

The NOTIFY response is generated as described in [2]. NOTIFY responses do not contain key-event Event message bodies.

16.9. Subscriber Generation of SUBSCRIBE Requests

The SUBSCRIBE request for "key-event" SIP events is generated as described in [2] with the following clarifications:

- the Event header is set to "Event: key-event".
- the Content-Type header is set to "Content-Type: application/key-event" if a Key-Event message body is included.
- if dialog specific subscription is required then the instance-header is added to the Key-Event message body as described in section 13.2.

The event-header will contain the list of desired key-events to receive notifications for and MAY include wildcards. The properties-header will contain the set of key-event properties for which the subscriber wishes to set or retrieved the value of.

A new "key-event" SUBSCRIBE request for an existing subscription dialog will entirely replace the subscription state of previous SUBSCRIBE requests in that dialog.

If the SIP Expires header has a value of zero then the subscriber application can still always expect at least one further NOTIFY request to be sent by the Notifier (as described in [2]).

When using the "key-event" package, if the SIP Expires header does not have a value of zero then the subscriber application MUST include a Key-Event message body.

16.10. Pre-loaded Routes in SUBSCRIBE requests

The following section applies only to dialog-specific subscriptions where the subscriber application is located along the associated INVITE dialog signaling path and wishes to collect key-events from one of the User Agent endpoints.

In order to maximize the likelihood that a SUBSCRIBE request will successfully reach the desired User Agent, it is RECOMMENDED the subscriber place pre-loaded Route headers in the SUBSCRIBE request to reproduce any Record-Routes established in the associated INVITE dialog. The presence of record-routed Application Level Gateways controlling firewalls and/or NATÆs is a typical example of when this may be helpful.

When the subscriber wishes to send a SUBSCRIBE request to the caller, the Route set is constructed (in the manner that the callee would normally use) from the Record-Route and Contact headers in the original INVITE request.

When the subscriber wishes to send a SUBSCRIBE request to the callee, the Route set is constructed (in the manner that the caller would normally use) from the Record-Route and Contact headers in the original 200 OK INVITE response. However in the case where the subscriber application is acting as a proxy in the original INVITE dialog (c.f., Usage Scenario 2, Section 2.2.2), then the subscriber should ignore all Record-Route headers up to and including the Record-Route inserted by the subscriber application in the 200 OK INVITE response.

16.11. Handling of Forked Subscriptions

A SUBSCRIBE request may fork and arrive at multiple devices. In this case, the subscriber can terminate those subscriptions it wishes by sending a SUBSCRIBE with an Expires value set to 0. It can also respond to any NOTIFYs from a UA with a 481 Transaction Does Not Exist.

If the subscriber wishes to accept multiple subscriptions, merging of state is not defined due to the fact that the multiple subscriptions represent the state of multiple devices.

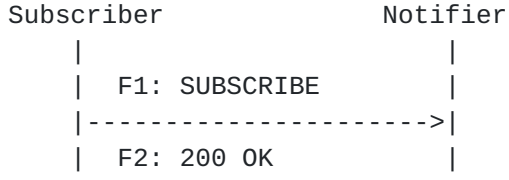
A Notifier SHOULD return 482 Request Merged response to subsequent multiple subscriptions having the same SUBSCRIBE request transaction id (even if they have differing request-uriÆs).

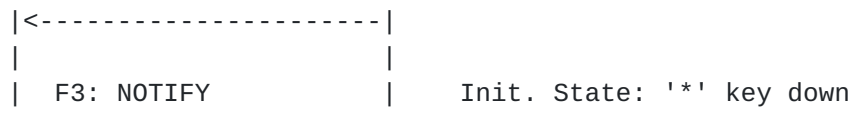
16.12. Rate of Notifications

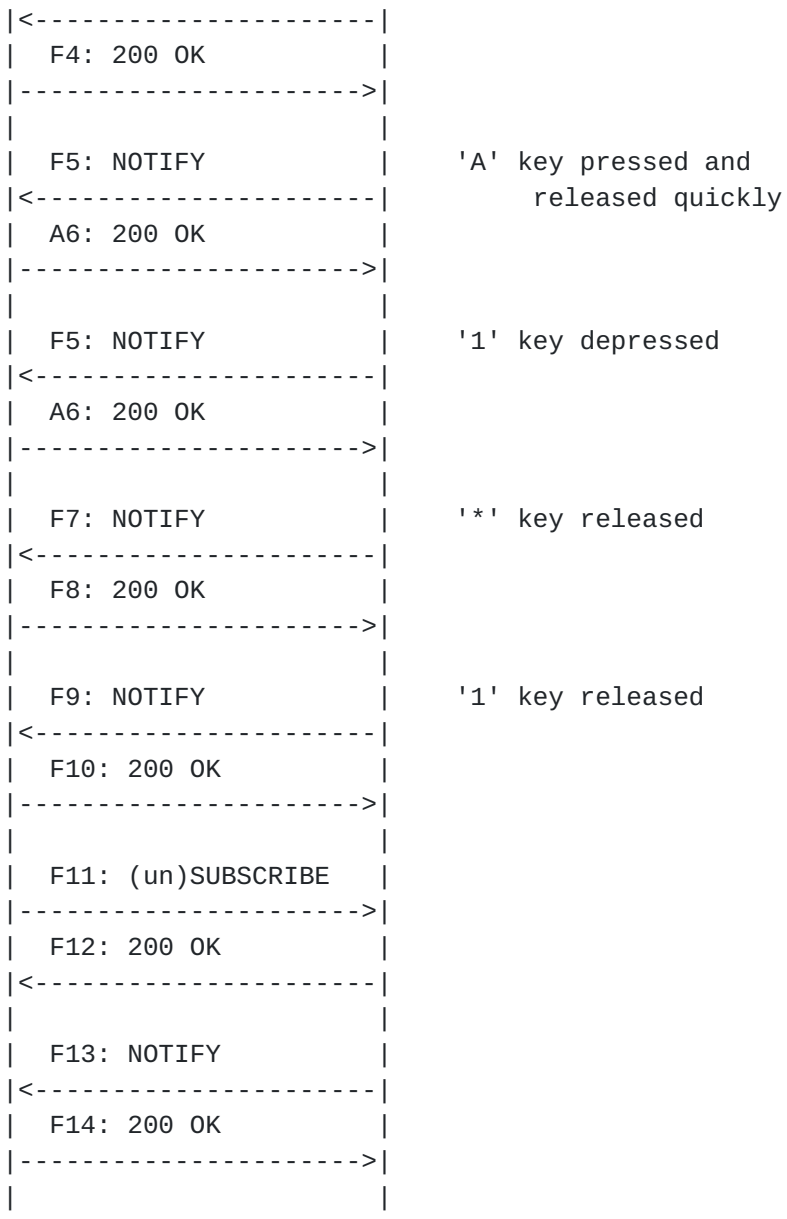
The use of the Key Events packages should be limited to situations that require limited amount of data to be transported. As each key press can cause a notification (and response) to be sent, this mechanism is inefficient in scenarios where a significant amount of data is to be transferred between two endpoints. However, as key event packages are designed to transport user indications, the rate of notifications should not be much more than ten per second nor more than 10 to 20 events at a time.

17. Examples

In the example call flow below, an application server subscribes to the status of a caller's keypad events. NOTIFY requests are sent for two key presses, in addition to the initial NOTIFY indicating those key-events the Notifier supports and their initial state. Via headers are omitted for clarity.




F3: NOTIFY
Init. State: '*' key down



F1: Subscriber -> Notifier

```
SUBSCRIBE sip:caller@access-22.isp.net SIP/2.0
To: <sip:caller@isp.net>
From: <sip:appl@appsrv.sp.com>;tag=2356
Call-Id: 321123@appsrv.sp.com
CSeq: 2 SUBSCRIBE
Contact: <sip:appl@appsrv.sp.com>
Event: key-event
Expires: 3600
Content: application/key-event
Content-Length: xx

t: 36539655 1
```

```
   i: call-id=8347-da8d-7657-ab32@192.144.22.1; local-tag=2342354556;
   remote-tag=00993k23ff8; attach=cspace
   e: key, keybd
   p: *.exists(), *.dwndly(1000)
```

The subscriber is requesting all key and keyboard sub-package key-events and also requesting to know the presence of multiple key instances and a delay in keydown notification of 1000ms.

F2: Notifier -> Subscriber

```
SIP/2.0 200 OK
To: <sip:caller@isp.net>;tag=789
From: <sip:appl@appsrv.sp.com>;tag=2356
Call-Id: 321123@appsrv.sp.com
CSeq: 2 SUBSCRIBE
Contact: <sip:caller@isp.net>
Expires: 3600
Content-Length: 0
```

F3: Notifier -> Subscriber

```
NOTIFY sip:appl@appsrv.sp.com SIP/2.0
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 403 NOTIFY
Contact: <sip:caller@isp.net>
Event: key-event
Content: application/key-event
Content-Length: xx

t: 36539658 1
e: key[@dialpad, 65-68]
p: key[@dialpad, 65-68].dwndly(1000)
s: key[42].keydown(-1902)
```

The Notifier confirms subscribed key-events and property values along with non-default initial key-event states. In this example the Notifier has created a subscription for keys { '0' - '9', '*', '#', 'A' - 'D' } and confirms that the dwndly property has been set. All keys except '*' are in the default (keyup) state; '*' has been depressed since 1.902 seconds before the specified NTP time value, that is, the key was depressed at 36539656.098 seconds past the epoch. No multiple key instances exist.

F4: Subscriber -> Notifier

```
SIP/2.0 200 OK
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 403 NOTIFY
Content-Length: 0
```

F5: Notifier -> Subscriber

```
NOTIFY sip:appl@appsrv.sp.com SIP/2.0
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 404 NOTIFY
Contact: <sip:caller@isp.net>
Event: key-event
Content: application/key-event
Content-Length: xx

t: 36539663 2
s: key[65].keyup(17, 537)
```

The Notifier reports that the 'A' key has been pressed and released. As the duration of depression was less that 1000ms, only the keyup state indication has been generated. In this case, the depression can be determined to have started at time 36539662.480 and lasted for 537ms.

F6: Subscriber -> Notifier

```
SIP/2.0 200 OK
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 404 NOTIFY
Content-Length: 0
```

F7: Notifier -> Subscriber

```
NOTIFY sip:appl@appsrv.sp.com SIP/2.0
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 405 NOTIFY
Contact: <sip:caller@isp.net>
Event: key-event
Content: application/key-event
Content-Length: xx

t: 36539668 2
s: key[49].keydown(-987)
```

The Notifier reports that the '1' key was pressed at time 36539667.13. The keydown has been generated because the key has been depressed for more than 1000ms. In this example the NOTIFY request was actually generated at time 36539668.13 which explains the -987ms offset from the NTP time.

F8: Subscriber -> Notifier

```
SIP/2.0 200 OK
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 405 NOTIFY
Content-Length: 0
```

F9: Notifier -> Subscriber

```
NOTIFY sip:appl@appsrv.sp.com SIP/2.0
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 406 NOTIFY
Contact: <sip:caller@isp.net>
Event: key-event
Content: application/key-event
Content-Length: xx

t: 36539669 3
s: key[42].keyup(756, 13658)
```

This notification indicates that the '*' key has eventually been released at time 36539669.756. The duration of depression is 13658 milliseconds; this value is consistent with the time of depression indicated in the initial (keydown) state indication of message F3.

F10: Subscriber -> Notifier

```
SIP/2.0 200 OK
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 406 NOTIFY
Content-Length: 0
```

F11: Notifier -> Subscriber

```
NOTIFY sip:appl@appsrv.sp.com SIP/2.0
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 407 NOTIFY
Contact: <sip:caller@isp.net>
Event: key-event
Content: application/key-event
Content-Length: xx
```

```
   t: 36539672 4
   e: key
   p: key[49].keyup(154, 5141)
```

This notification indicates that the '1' key has been released at time 36539672.154. The duration of depression is 5141 milliseconds; this value is consistent with the time of depression indicated in the (keydown) state indication of message F7.

F12: Subscriber -> Notifier

```
SIP/2.0 200 OK
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 407 NOTIFY
Content-Length: 0
```

F13: Subscriber -> Notifier

```
SUBSCRIBE sip:caller@access-22.isp.net SIP/2.0
To: <sip:caller@isp.net>;tag=789
From: <sip:appl@appsrv.sp.com>;tag=2356
Call-Id: 321123@appsrv.sp.com
CSeq: 3 SUBSCRIBE
Contact: <sip:appl@appsrv.sp.com>
Event: key-event
Expires: 0
Content-Length: 0
```

The subscriber indicates it wishes to terminate the subscription.

F14: Notifier -> Subscriber

```
SIP/2.0 200 OK
To: <sip:caller@isp.net>;tag=789
From: <sip:appl@appsrv.sp.com>;tag=2356
Call-Id: 321123@appsrv.sp.com
CSeq: 3 SUBSCRIBE
Contact: <sip:caller@isp.net>
Expires: 0
Content-Length: 0
```

F15: Notifier -> Subscriber

```
NOTIFY sip:appl@appsrv.sp.com SIP/2.0
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 408 NOTIFY
Contact: <sip:caller@isp.net>
```

```
   Subscription-Expires: 0
   Content-Length: 0
```

As described in [1], the Notifier must always send one NOTIFY request in response to any accepted SUBSCRIBE request.

F16: Subscriber -> Notifier

```
SIP/2.0 200 OK
To: <sip:appl@appsrv.sp.com>;tag=2356
From: <sip:caller@isp.net>;tag=789
Call-Id: 321123@appsrv.sp.com
CSeq: 408 NOTIFY
Content-Length: 0
```

18. Security Considerations

Key-Event Subscriptions are very likely to reveal sensitive information such as pin-numbers and destination numbers. Therefore, it is REQUIRED that devices accepting SUBSCRIBE requests perform some level of authentication before establishing a subscription. When such authentication is not provided by the network layer (e.g., IPSEC or private/trusted networks), then it MUST be provided by the transport-layer (e.g., TLS) or application-layer (e.g., SIP message/header authentication).

Likewise, the information provided in the NOTIFY request is also likely to contain sensitive information and so the Notifier MUST ensure that the NOTIFY requests are transported securely. Once again, this protection could be provided by the network-layer (e.g., IPSEC encryption or private networks), transport layer (e.g., TLS encryption) or application layer (e.g., S/MIME or SIP message body encryption).

Alternatively, it is possible for the Notifier to allow unsecured/unauthenticated subscribers to subscribe to key-events that will not divulge sensitive information. For instance, subscriptions to the "*" and "#" keys should not divulge sensitive information but may provide sufficient functionality for some 3pcc/B2BUA pre-paid calling card applications.

19. IANA Considerations

Section 12 provides the registration information needed to register the "application/key-event" MIME type with IANA.

19.1. Key-Event Package Registration

This document specifies an event package as defined in [2]. The following information is specified as required by "SIP-Specific Event Notification" [RFC xxxx].

```
   Package Name: key-events
   Type: package
```

Contact: Robert Fairlie-Cuninghame <rfairlie@nuera.com>, Bert Culpepper <bert.culpepper@intervoice-brite.com>, Jean-Francois Mule <jf.mule@cablelabs.com>

19.2. Key-Event Sub-Package Registration

This document defines a Key-Event Sub-Package namespace that should be maintained by a centralized coordinating organization. Sub-Package names share the same namespace. The following Key-Event Sub-Package registration information is specified in anticipation of the mechanisms and namespace defined in this document being accepted in the IETF.

Sub-Package Name	Contact	Reference
Key	[RFC,BC,JFM]	[RFC xxxx]
Keyboard	[RFC,BC,JFM]	[RFC xxxx]
Telephone	[RFC,BC,JFM]	[RFC xxxx]
User	[RFC,BC,JFM]	[RFC xxxx]

People:
[RFC] Robert Fairlie-Cuninghame <rfairlie@nuera.com>
[BC] Bert Culpepper <bert.culpepper@intervoice-brite.com>
[JFM] Jean-Francois Mule <jf.mule@cablelabs.com>

References:
[RFC xxxx] B. Culpepper, R. Fairlie-Cuninghame, J. Mule, "SIP Event Package for Keys", March 2002.

Guidelines for registering key event packages with IANA will be completed in a future draft revision.

20. Authors

Robert Fairlie-Cuninghame
Nuera Communications, Inc.
50 Victoria Rd
Farnborough, Hants GU14-7PG
United Kingdom
Phone: +44-1252-548200
Email: rfairlie@nuera.com

Bert Culpepper
InterVoice-Brite, Inc.
701 International Parkway
Heathrow, FL 32746
Phone: 407-357-1536
Email: bert.culpepper@intervoice-brite.com

Jean-Francois Mule

Cable Television Laboratories, Inc.
400 Centennial Parkway
Louisville, CO 80027-1266
Phone: 303-661-3708
Email: jf.mule@cablelabs.com

21. References

1 S. Bradner, "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.

2 A. Roach, "SIP-Specific Event Notification", Internet-Draft draft-ietf-sip-events-04, February 2002, Work in progress.

3 M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: Session Initiation Protocol", RFC 2543, March 1999.

4 H. Schulzrinne, S. Petrack, "RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals", RFC 2833, May 2000.

5 R. Mahy, "A Call Control Model for SIP", Internet-Draft, November 2001, Work in progress.

6 D. Mills, "Network Time Protocol (Version 3)", RFC1305, March 1992.

7 The Unicode Consortium, "The Unicode Standard -- Version 3.0", ISBN 0-201-61633-5. Described at http://www.unicode.org/unicode/standard/versions/Unicode3.0.html