           Protocol Independent Multicast - Sparse Mode (PIM-SM):
                      Protocol Specification (Revised)

Status of This Memo

   This document specifies an Internet standards track protocol for the
   Internet community, and requests discussion and suggestions for
   improvements. Please refer to the current edition of the "Internet
   Official Protocol Standards" (STD 1) for the standardization state
   and status of this protocol. Distribution of this memo is unlimited.

Abstract

   This document specifies Protocol Independent Multicast - Sparse Mode
   (PIM-SM). PIM-SM is a multicast routing protocol that can use the
   underlying unicast routing information base or a separate multicast-
   capable routing information base. It builds unidirectional shared
   trees rooted at a Rendezvous Point (RP) per group, and optionally
   creates shortest-path trees per source.

   This document obsoletes RFC 2362, an Experimental version of PIM-SM.

Table of Contents

List of Figures

1. Introduction

   This document specifies a protocol for efficiently routing multicast
   groups that may span wide-area (and inter-domain) internets. This
   protocol is called Protocol Independent Multicast - Sparse Mode
   (PIM-SM) because, although it may use the underlying unicast routing
   to provide reverse-path information for multicast tree building, it
   is not dependent on any particular unicast routing protocol.

   PIM-SM version 2 was originally specified in RFC 2117 and was revised
   in RFC 2362, both Experimental RFCs. This document is intended to
   obsolete RFC 2362, to correct a number of deficiencies that have been
   identified with the way PIM-SM was previously specified, and to bring
   PIM-SM onto the IETF Standards Track. As far as possible, this
   document specifies the same protocol as RFC 2362 and only diverges
   from the behavior intended by RFC 2362 when the previously specified
   behavior was clearly incorrect. Routers implemented according to the
   specification in this document will be able to interoperate
   successfully with routers implemented according to RFC 2362.

2. Terminology

   In this document, the key words "MUST", "MUST NOT", "REQUIRED",
   "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY",
   and "OPTIONAL" are to be interpreted as described in RFC 2119 [1] and
   indicate requirement levels for compliant PIM-SM implementations.

2.1. Definitions

   The following terms have special significance for PIM-SM:

   Rendezvous Point (RP):
         An RP is a router that has been configured to be used as the
         root of the non-source-specific distribution tree for a
         multicast group. Join messages from receivers for a group are
         sent towards the RP, and data from senders is sent to the RP so
         that receivers can discover who the senders are and start to
         receive traffic destined for the group.

   Designated Router (DR):
         A shared-media LAN like Ethernet may have multiple PIM-SM
         routers connected to it. A single one of these routers, the
         DR, will act on behalf of directly connected hosts with respect
         to the PIM-SM protocol. A single DR is elected per interface
         (LAN or otherwise) using a simple election process.

MRIB Multicast Routing Information Base. This is the multicast
      topology table, which is typically derived from the unicast
      routing table, or routing protocols such as Multiprotocol BGP
      (MBGP) that carry multicast-specific topology information. In
      PIM-SM, the MRIB is used to decide where to send Join/Prune
      messages. A secondary function of the MRIB is to provide
      routing metrics for destination addresses; these metrics are
      used when sending and processing Assert messages.

RPF Neighbor
      RPF stands for "Reverse Path Forwarding". The RPF Neighbor of
      a router with respect to an address is the neighbor that the
      MRIB indicates should be used to forward packets to that
      address. In the case of a PIM-SM multicast group, the RPF
      neighbor is the router that a Join message for that group would
      be directed to, in the absence of modifying Assert state.

TIB   Tree Information Base. This is the collection of state at a
      PIM router that has been created by receiving PIM Join/Prune
      messages, PIM Assert messages, and Internet Group Management
      Protocol (IGMP) or Multicast Listener Discovery (MLD)
      information from local hosts. It essentially stores the state
      of all multicast distribution trees at that router.

MFIB Multicast Forwarding Information Base. The TIB holds all the
      state that is necessary to forward multicast packets at a
      router. However, although this specification defines
      forwarding in terms of the TIB, to actually forward packets
      using the TIB is very inefficient. Instead, a real router
      implementation will normally build an efficient MFIB from the
      TIB state to perform forwarding. How this is done is
      implementation-specific and is not discussed in this document.

Upstream
      Towards the root of the tree. The root of tree may be either
      the source or the RP, depending on the context.

Downstream
      Away from the root of the tree.

GenID Generation Identifier, used to detect reboots.

PMBR PIM Multicast Border Router, joining a PIM domain with another
      multicast domain.

2.2. Pseudocode Notation

   We use set notation in several places in this specification.

   A (+) B is the union of two sets, A and B.

   A (-) B is the elements of set A that are not in set B.

   NULL   is the empty set or list.

   In addition, we use C-like syntax:

   =   denotes assignment of a variable.

   ==   denotes a comparison for equality.

   !=   denotes a comparison for inequality.

   Braces { and } are used for grouping.

3. PIM-SM Protocol Overview

   This section provides an overview of PIM-SM behavior. It is intended
   as an introduction to how PIM-SM works, and it is NOT definitive.
   For the definitive specification, see Section 4.

   PIM relies on an underlying topology-gathering protocol to populate a
   routing table with routes. This routing table is called the
   Multicast Routing Information Base (MRIB). The routes in this table
   may be taken directly from the unicast routing table, or they may be
   different and provided by a separate routing protocol such as MBGP
   [10]. Regardless of how it is created, the primary role of the MRIB
   in the PIM protocol is to provide the next-hop router along a
   multicast-capable path to each destination subnet. The MRIB is used
   to determine the next-hop neighbor to which any PIM Join/Prune
   message is sent. Data flows along the reverse path of the Join
   messages. Thus, in contrast to the unicast RIB, which specifies the
   next hop that a data packet would take to get to some subnet, the
   MRIB gives reverse-path information and indicates the path that a
   multicast data packet would take from its origin subnet to the router
   that has the MRIB.

   Like all multicast routing protocols that implement the service model
   from RFC 1112 [3], PIM-SM must be able to route data packets from
   sources to receivers without either the sources or receivers knowing
   a priori of the existence of the others. This is essentially done in
   three phases, although as senders and receivers may come and go at
   any time, all three phases may occur simultaneously.

3.1. Phase One: RP Tree

   In phase one, a multicast receiver expresses its interest in
   receiving traffic destined for a multicast group. Typically, it does
   this using IGMP [2] or MLD [4], but other mechanisms might also serve
   this purpose. One of the receiver's local routers is elected as the
   Designated Router (DR) for that subnet. On receiving the receiver's
   expression of interest, the DR then sends a PIM Join message towards
   the RP for that multicast group. This Join message is known as a
   (*,G) Join because it joins group G for all sources to that group.
   The (*,G) Join travels hop-by-hop towards the RP for the group, and
   in each router it passes through, multicast tree state for group G is
   instantiated. Eventually, the (*,G) Join either reaches the RP or
   reaches a router that already has (*,G) Join state for that group.
   When many receivers join the group, their Join messages converge on
   the RP and form a distribution tree for group G that is rooted at the
   RP. This is known as the RP Tree (RPT), and is also known as the
   shared tree because it is shared by all sources sending to that
   group. Join messages are resent periodically so long as the receiver
   remains in the group. When all receivers on a leaf-network leave the
   group, the DR will send a PIM (*,G) Prune message towards the RP for
   that multicast group. However, if the Prune message is not sent for
   any reason, the state will eventually time out.

   A multicast data sender just starts sending data destined for a
   multicast group. The sender's local router (DR) takes those data
   packets, unicast-encapsulates them, and sends them directly to the
   RP. The RP receives these encapsulated data packets, decapsulates
   them, and forwards them onto the shared tree. The packets then
   follow the (*,G) multicast tree state in the routers on the RP Tree,
   being replicated wherever the RP Tree branches, and eventually
   reaching all the receivers for that multicast group. The process of
   encapsulating data packets to the RP is called registering, and the
   encapsulation packets are known as PIM Register packets.

   At the end of phase one, multicast traffic is flowing encapsulated to
   the RP, and then natively over the RP tree to the multicast
   receivers.

3.2. Phase Two: Register-Stop

   Register-encapsulation of data packets is inefficient for two
   reasons:

   o Encapsulation and decapsulation may be relatively expensive
     operations for a router to perform, depending on whether or not the
     router has appropriate hardware for these tasks.

   o Traveling all the way to the RP, and then back down the shared tree
     may result in the packets traveling a relatively long distance to
     reach receivers that are close to the sender. For some
     applications, this increased latency or bandwidth consumption is
     undesirable.

   Although Register-encapsulation may continue indefinitely, for these
   reasons, the RP will normally choose to switch to native forwarding.
   To do this, when the RP receives a register-encapsulated data packet
   from source S on group G, it will normally initiate an (S,G) source-
   specific Join towards S. This Join message travels hop-by-hop
   towards S, instantiating (S,G) multicast tree state in the routers
   along the path. (S,G) multicast tree state is used only to forward
   packets for group G if those packets come from source S. Eventually
   the Join message reaches S's subnet or a router that already has
   (S,G) multicast tree state, and then packets from S start to flow
   following the (S,G) tree state towards the RP. These data packets
   may also reach routers with (*,G) state along the path towards the
   RP; if they do, they can shortcut onto the RP tree at this point.

   While the RP is in the process of joining the source-specific tree
   for S, the data packets will continue being encapsulated to the RP.
   When packets from S also start to arrive natively at the RP, the RP
   will be receiving two copies of each of these packets. At this
   point, the RP starts to discard the encapsulated copy of these
   packets, and it sends a Register-Stop message back to S's DR to
   prevent the DR from unnecessarily encapsulating the packets.

   At the end of phase 2, traffic will be flowing natively from S along
   a source-specific tree to the RP, and from there along the shared
   tree to the receivers. Where the two trees intersect, traffic may
   transfer from the source-specific tree to the RP tree and thus avoid
   taking a long detour via the RP.

   Note that a sender may start sending before or after a receiver joins
   the group, and thus phase two may happen before the shared tree to
   the receiver is built.

3.3. Phase Three: Shortest-Path Tree

   Although having the RP join back towards the source removes the
   encapsulation overhead, it does not completely optimize the
   forwarding paths. For many receivers, the route via the RP may
   involve a significant detour when compared with the shortest path
   from the source to the receiver.

   To obtain lower latencies or more efficient bandwidth utilization, a
   router on the receiver's LAN, typically the DR, may optionally
   initiate a transfer from the shared tree to a source-specific
   shortest-path tree (SPT). To do this, it issues an (S,G) Join
   towards S. This instantiates state in the routers along the path to
   S. Eventually, this join either reaches S's subnet or reaches a
   router that already has (S,G) state. When this happens, data packets
   from S start to flow following the (S,G) state until they reach the
   receiver.

   At this point, the receiver (or a router upstream of the receiver)
   will be receiving two copies of the data: one from the SPT and one
   from the RPT. When the first traffic starts to arrive from the SPT,
   the DR or upstream router starts to drop the packets for G from S
   that arrive via the RP tree. In addition, it sends an (S,G) Prune
   message towards the RP. This is known as an (S,G,rpt) Prune. The
   Prune message travels hop-by-hop, instantiating state along the path
   towards the RP indicating that traffic from S for G should NOT be
   forwarded in this direction. The prune is propagated until it
   reaches the RP or a router that still needs the traffic from S for
   other receivers.

   By now, the receiver will be receiving traffic from S along the
   shortest-path tree between the receiver and S. In addition, the RP
   is receiving the traffic from S, but this traffic is no longer
   reaching the receiver along the RP tree. As far as the receiver is
   concerned, this is the final distribution tree.

3.4. Source-Specific Joins

   IGMPv3 permits a receiver to join a group and specify that it only
   wants to receive traffic for a group if that traffic comes from a
   particular source. If a receiver does this, and no other receiver on
   the LAN requires all the traffic for the group, then the DR may omit
   performing a (*,G) join to set up the shared tree, and instead issue
   a source-specific (S,G) join only.

   The range of multicast addresses from 232.0.0.0 to 232.255.255.255 is
   currently set aside for source-specific multicast in IPv4. For
   groups in this range, receivers should only issue source-specific
   IGMPv3 joins. If a PIM router receives a non-source-specific join
   for a group in this range, it should ignore it, as described in
   Section 4.8.

3.5. Source-Specific Prunes

   IGMPv3 also permits a receiver to join a group and to specify that it
   only wants to receive traffic for a group if that traffic does not
   come from a specific source or sources. In this case, the DR will
   perform a (*,G) join as normal, but may combine this with an
   (S,G,rpt) prune for each of the sources the receiver does not wish to
   receive.

3.6. Multi-Access Transit LANs

   The overview so far has concerned itself with point-to-point transit
   links. However, using multi-access LANs such as Ethernet for transit
   is not uncommon. This can cause complications for three reasons:

   o Two or more routers on the LAN may issue (*,G) Joins to different
     upstream routers on the LAN because they have inconsistent MRIB
     entries regarding how to reach the RP. Both paths on the RP tree
     will be set up, causing two copies of all the shared tree traffic
     to appear on the LAN.

   o Two or more routers on the LAN may issue (S,G) Joins to different
     upstream routers on the LAN because they have inconsistent MRIB
     entries regarding how to reach source S. Both paths on the source-
     specific tree will be set up, causing two copies of all the traffic
     from S to appear on the LAN.

   o A router on the LAN may issue a (*,G) Join to one upstream router
     on the LAN, and another router on the LAN may issue an (S,G) Join
     to a different upstream router on the same LAN. Traffic from S may
     reach the LAN over both the RPT and the SPT. If the receiver
     behind the downstream (*,G) router doesn't issue an (S,G,rpt)
     prune, then this condition would persist.

   All of these problems are caused by there being more than one
   upstream router with join state for the group or source-group pair.
   PIM does not prevent such duplicate joins from occurring; instead,
   when duplicate data packets appear on the LAN from different routers,
   these routers notice this and then elect a single forwarder. This
   election is performed using PIM Assert messages, which resolve the
   problem in favor of the upstream router that has (S,G) state; or, if
   neither or both router has (S,G) state, then the problem is resolved
   in favor of the router with the best metric to the RP for RP trees,
   or the best metric to the source to source-specific trees.

   These Assert messages are also received by the downstream routers on
   the LAN, and these cause subsequent Join messages to be sent to the
   upstream router that won the Assert.

3.7. RP Discovery

   PIM-SM routers need to know the address of the RP for each group for
   which they have (*,G) state. This address is obtained automatically
   (e.g., embedded-RP), through a bootstrap mechanism, or through static
   configuration.

   One dynamic way to do this is to use the Bootstrap Router (BSR)
   mechanism [11]. One router in each PIM domain is elected the
   Bootstrap Router through a simple election process. All the routers
   in the domain that are configured to be candidates to be RPs
   periodically unicast their candidacy to the BSR. From the
   candidates, the BSR picks an RP-set, and periodically announces this
   set in a Bootstrap message. Bootstrap messages are flooded hop-by-
   hop throughout the domain until all routers in the domain know the
   RP-Set.

   To map a group to an RP, a router hashes the group address into the
   RP-set using an order-preserving hash function (one that minimizes
   changes if the RP-Set changes). The resulting RP is the one that it
   uses as the RP for that group.

4. Protocol Specification

   The specification of PIM-SM is broken into several parts:

   o Section 4.1 details the protocol state stored.

   o Section 4.2 specifies the data packet forwarding rules.

   o Section 4.3 specifies Designated Router (DR) election and the rules
     for sending and processing Hello messages.

   o Section 4.4 specifies the PIM Register generation and processing
     rules.

   o Section 4.5 specifies the PIM Join/Prune generation and processing
     rules.

   o Section 4.6 specifies the PIM Assert generation and processing
     rules.

   o Section 4.7 specifies the RP discovery mechanisms.

   o The subset of PIM required to support Source-Specific Multicast,
     PIM-SSM, is described in Section 4.8.

   o PIM packet formats are specified in Section 4.9.

      o A summary of PIM-SM timers and their default values is given in
        Section 4.10.

      o Appendix A specifies the PIM Multicast Border Router behavior.

   4.1. PIM Protocol State

      This section specifies all the protocol state that a PIM
      implementation should maintain in order to function correctly. We
      term this state the Tree Information Base (TIB), as it holds the
      state of all the multicast distribution trees at this router. In
      this specification, we define PIM mechanisms in terms of the TIB.
      However, only a very simple implementation would actually implement
      packet forwarding operations in terms of this state. Most
      implementations will use this state to build a multicast forwarding
      table, which would then be updated when the relevant state in the TIB
      changes.

      Although we specify precisely the state to be kept, this does not
      mean that an implementation of PIM-SM needs to hold the state in this
      form. This is actually an abstract state definition, which is needed
      in order to specify the router's behavior. A PIM-SM implementation
      is free to hold whatever internal state it requires and will still be
      conformant with this specification so long as it results in the same
      externally visible protocol behavior as an abstract router that holds
      the following state.

      We divide TIB state into four sections:

      (*,*,RP) state
           State that maintains per-RP trees, for all groups served by a
           given RP.

      (*,G) state
           State that maintains the RP tree for G.

      (S,G) state
           State that maintains a source-specific tree for source S and
           group G.

      (S,G,rpt) state
           State that maintains source-specific information about source S
           on the RP tree for G. For example, if a source is being
           received on the source-specific tree, it will normally have been
           pruned off the RP tree. This prune state is (S,G,rpt) state.

The state that should be kept is described below. Of course,
implementations will only maintain state when it is relevant to
forwarding operations; for example, the "NoInfo" state might be
assumed from the lack of other state information rather than being
held explicitly.

4.1.1. General Purpose State

A router holds the following non-group-specific state:

For each interface:

> o Effective Override Interval
>
> o Effective Propagation Delay
>
> o Suppression state: One of {"Enable", "Disable"}
>
> Neighbor State:
>
>> For each neighbor:
>>
>>> o Information from neighbor's Hello
>>>
>>> o Neighbor's GenID.
>>>
>>> o Neighbor Liveness Timer (NLT)
>>
>> Designated Router (DR) State:
>>
>>> o Designated Router's IP Address
>>>
>>> o DR's DR Priority

The Effective Override Interval, the Effective Propagation Delay and
the Interface suppression state are described in Section 4.3.3.
Designated Router state is described in Section 4.3.

4.1.2. (*,*,RP) State

   For every RP, a router keeps the following state:

   (*,*,RP) state:
        For each interface:

             PIM (*,*,RP) Join/Prune State:

                  o State: One of {"NoInfo" (NI), "Join" (J), "Prune-
                    Pending" (PP)}

                  o Prune-Pending Timer (PPT)

                  o Join/Prune Expiry Timer (ET)

        Not interface specific:

             Upstream (*,*,RP) Join/Prune State:

                  o State: One of {"NotJoined(*,*,RP)",
                    "Joined(*,*,RP)"}

                  o Upstream Join/Prune Timer (JT)

                  o Last RPF Neighbor towards RP that was used

   PIM (*,*,RP) Join/Prune state is the result of receiving PIM (*,*,RP)
   Join/Prune messages on this interface and is specified in Section
   4.5.1.

   The upstream (*,*,RP) Join/Prune State reflects the state of the
   upstream (*,*,RP) state machine described in Section 4.5.5.

   The upstream (*,*,RP) Join/Prune Timer is used to send out periodic
   Join(*,*,RP) messages, and to override Prune(*,*,RP) messages from
   peers on an upstream LAN interface.

   The last RPF neighbor towards the RP is stored because if the MRIB
   changes, then the RPF neighbor towards the RP may change. If it does
   so, then we need to trigger a new Join(*,*,RP) to the new upstream
   neighbor and a Prune(*,*,RP) to the old upstream neighbor.
   Similarly, if a router detects through a changed GenID in a Hello
   message that the upstream neighbor towards the RP has rebooted, then
   it should re-instantiate state by sending a Join(*,*,RP). These
   mechanisms are specified in Section 4.5.5.

4.1.3. (*,G) State

   For every group G, a router keeps the following state:

   (*,G) state:
        For each interface:

              Local Membership:
                   State: One of {"NoInfo", "Include"}

              PIM (*,G) Join/Prune State:

                   o State: One of {"NoInfo" (NI), "Join" (J), "Prune-
                     Pending" (PP)}

                   o Prune-Pending Timer (PPT)

                   o Join/Prune Expiry Timer (ET)

              (*,G) Assert Winner State

                   o State: One of {"NoInfo" (NI), "I lost Assert" (L),
                     "I won Assert" (W)}

                   o Assert Timer (AT)

                   o Assert winner's IP Address (AssertWinner)

                   o Assert winner's Assert Metric (AssertWinnerMetric)

          Not interface specific:

              Upstream (*,G) Join/Prune State:

                   o State: One of {"NotJoined(*,G)", "Joined(*,G)"}

              o Upstream Join/Prune Timer (JT)

              o Last RP Used

              o Last RPF Neighbor towards RP that was used

   Local membership is the result of the local membership mechanism
   (such as IGMP or MLD) running on that interface. It need not be kept
   if this router is not the DR on that interface unless this router won
   a (*,G) assert on this interface for this group, although
   implementations may optionally keep this state in case they become
   the DR or assert winner. We recommend storing this information if

possible, as it reduces latency converging to stable operating
conditions after a failure causing a change of DR. This information
is used by the pim_include(*,G) macro described in Section 4.1.6.

PIM (*,G) Join/Prune state is the result of receiving PIM (*,G)
Join/Prune messages on this interface and is specified in Section
4.5.2. The state is used by the macros that calculate the outgoing
interface list in Section 4.1.6, and in the JoinDesired(*,G) macro
(defined in Section 4.5.6) that is used in deciding whether a
Join(*,G) should be sent upstream.

(*,G) Assert Winner state is the result of sending or receiving (*,G)
Assert messages on this interface. It is specified in Section 4.6.2.

The upstream (*,G) Join/Prune State reflects the state of the
upstream (*,G) state machine described in Section 4.5.6.

The upstream (*,G) Join/Prune Timer is used to send out periodic
Join(*,G) messages, and to override Prune(*,G) messages from peers on
an upstream LAN interface.

The last RP used must be stored because if the RP-Set changes
(Section 4.7), then state must be torn down and rebuilt for groups
whose RP changes.

The last RPF neighbor towards the RP is stored because if the MRIB
changes, then the RPF neighbor towards the RP may change. If it does
so, then we need to trigger a new Join(*,G) to the new upstream
neighbor and a Prune(*,G) to the old upstream neighbor. Similarly,
if a router detects through a changed GenID in a Hello message that
the upstream neighbor towards the RP has rebooted, then it should
re-instantiate state by sending a Join(*,G). These mechanisms are
specified in Section 4.5.6.

4.1.4. (S,G) State

   For every source/group pair (S,G), a router keeps the following
   state:

   (S,G) state:

        For each interface:

             Local Membership:
                  State: One of {"NoInfo", "Include"}

PIM (S,G) Join/Prune State:

    o State: One of {"NoInfo" (NI), "Join" (J), "Prune-
      Pending" (PP)}

    o Prune-Pending Timer (PPT)

    o Join/Prune Expiry Timer (ET)

(S,G) Assert Winner State

    o State: One of {"NoInfo" (NI), "I lost Assert" (L),
      "I won Assert" (W)}

    o Assert Timer (AT)

    o Assert winner's IP Address (AssertWinner)

    o Assert winner's Assert Metric (AssertWinnerMetric)

Not interface specific:

Upstream (S,G) Join/Prune State:

    o State: One of {"NotJoined(S,G)", "Joined(S,G)"}

o Upstream (S,G) Join/Prune Timer (JT)

o Last RPF Neighbor towards S that was used

o SPTbit (indicates (S,G) state is active)

o (S,G) Keepalive Timer (KAT)

Additional (S,G) state at the DR:

    o Register state: One of {"Join" (J), "Prune" (P),
      "Join-Pending" (JP), "NoInfo" (NI)}

    o Register-Stop timer

Additional (S,G) state at the RP:

    o PMBR: the first PMBR to send a Register for this
      source with the Border bit set.

Local membership is the result of the local source-specific
membership mechanism (such as IGMP version 3) running on that
interface and specifying that this particular source should be
included. As stored here, this state is the resulting state after
any IGMPv3 inconsistencies have been resolved. It need not be kept
if this router is not the DR on that interface unless this router won
a (S,G) assert on this interface for this group. However, we
recommend storing this information if possible, as it reduces latency
converging to stable operating conditions after a failure causing a
change of DR. This information is used by the pim_include(S,G) macro
described in Section 4.1.6.

PIM (S,G) Join/Prune state is the result of receiving PIM (S,G)
Join/Prune messages on this interface and is specified in Section
4.5.2. The state is used by the macros that calculate the outgoing
interface list in Section 4.1.6, and in the JoinDesired(S,G) macro
(defined in Section 4.5.7) that is used in deciding whether a
Join(S,G) should be sent upstream.

(S,G) Assert Winner state is the result of sending or receiving (S,G)
Assert messages on this interface. It is specified in Section 4.6.1.

The upstream (S,G) Join/Prune State reflects the state of the
upstream (S,G) state machine described in Section 4.5.7.

The upstream (S,G) Join/Prune Timer is used to send out periodic
Join(S,G) messages, and to override Prune(S,G) messages from peers on
an upstream LAN interface.

The last RPF neighbor towards S is stored because if the MRIB
changes, then the RPF neighbor towards S may change. If it does so,
then we need to trigger a new Join(S,G) to the new upstream neighbor
and a Prune(S,G) to the old upstream neighbor. Similarly, if the
router detects through a changed GenID in a Hello message that the
upstream neighbor towards S has rebooted, then it should re-
instantiate state by sending a Join(S,G). These mechanisms are
specified in Section 4.5.7.

The SPTbit is used to indicate whether forwarding is taking place on
the (S,G) Shortest Path Tree (SPT) or on the (*,G) tree. A router
can have (S,G) state and still be forwarding on (*,G) state during
the interval when the source-specific tree is being constructed.
When SPTbit is FALSE, only (*,G) forwarding state is used to forward
packets from S to G. When SPTbit is TRUE, both (*,G) and (S,G)
forwarding state are used.

The (S,G) Keepalive Timer is updated by data being forwarded using
this (S,G) forwarding state. It is used to keep (S,G) state alive in
the absence of explicit (S,G) Joins. Amongst other things, this is
necessary for the so-called "turnaround rules" -- when the RP uses
(S,G) joins to stop encapsulation, and then (S,G) prunes to prevent
traffic from unnecessarily reaching the RP.

On a DR, the (S,G) Register State is used to keep track of whether to
encapsulate data to the RP on the Register Tunnel; the (S,G)
Register-Stop timer tracks how long before encapsulation begins again
for a given (S,G).

On an RP, the PMBR value must be cleared when the Keepalive Timer
expires.

4.1.5. (S,G,rpt) State

For every source/group pair (S,G) for which a router also has (*,G)
state, it also keeps the following state:

(S,G,rpt) state:

     For each interface:

          Local Membership:
               State: One of {"NoInfo", "Exclude"}

          PIM (S,G,rpt) Join/Prune State:

               o State: One of {"NoInfo", "Pruned", "Prune-
                 Pending"}

               o Prune-Pending Timer (PPT)

               o Join/Prune Expiry Timer (ET)

     Not interface specific:

          Upstream (S,G,rpt) Join/Prune State:

               o State: One of {"RPTNotJoined(G)",
                 "NotPruned(S,G,rpt)", "Pruned(S,G,rpt)"}

               o Override Timer (OT)

Local membership is the result of the local source-specific
membership mechanism (such as IGMPv3) running on that interface and
specifying that although there is (*,G) Include state, this

particular source should be excluded. As stored here, this state is
the resulting state after any IGMPv3 inconsistencies between LAN
members have been resolved. It need not be kept if this router is
not the DR on that interface unless this router won a (*,G) assert on
this interface for this group. However, we recommend storing this
information if possible, as it reduces latency converging to stable
operating conditions after a failure causing a change of DR. This
information is used by the pim_exclude(S,G) macro described in
Section 4.1.6.

PIM (S,G,rpt) Join/Prune state is the result of receiving PIM
(S,G,rpt) Join/Prune messages on this interface and is specified in
Section 4.5.4. The state is used by the macros that calculate the
outgoing interface list in Section 4.1.6, and in the rules for adding
Prune(S,G,rpt) messages to Join(*,G) messages specified in Section
4.5.8.

The upstream (S,G,rpt) Join/Prune state is used along with the
Override Timer to send the correct override messages in response to
Join/Prune messages sent by upstream peers on a LAN. This state and
behavior are specified in Section 4.5.9.

4.1.6. State Summarization Macros

Using this state, we define the following "macro" definitions, which
we will use in the descriptions of the state machines and pseudocode
in the following sections.

The most important macros are those that define the outgoing
interface list (or "olist") for the relevant state. An olist can be
"immediate" if it is built directly from the state of the relevant
type. For example, the immediate_olist(S,G) is the olist that would
be built if the router only had (S,G) state and no (*,G) or (S,G,rpt)
state. In contrast, the "inherited" olist inherits state from other
types. For example, the inherited_olist(S,G) is the olist that is
relevant for forwarding a packet from S to G using both source-
specific and group-specific state.

There is no immediate_olist(S,G,rpt) as (S,G,rpt) state is negative
state; it removes interfaces in the (*,G) olist from the olist that
is actually used to forward traffic. The inherited_olist(S,G,rpt) is
therefore the olist that would be used for a packet from S to G
forwarding on the RP tree. It is a strict subset of
(immediate_olist(*,*,RP) (+) immediate_olist(*,G)).

Generally speaking, the inherited olists are used for forwarding, and
the immediate_olists are used to make decisions about state
maintenance.

```
immediate_olist(*,*,RP) =
    joins(*,*,RP)

immediate_olist(*,G) =
    joins(*,G) (+) pim_include(*,G) (-) lost_assert(*,G)

immediate_olist(S,G) =
    joins(S,G) (+) pim_include(S,G) (-) lost_assert(S,G)

inherited_olist(S,G,rpt) =
        ( joins(*,*,RP(G)) (+) joins(*,G) (-) prunes(S,G,rpt) )
    (+) ( pim_include(*,G) (-) pim_exclude(S,G))
    (-) ( lost_assert(*,G) (+) lost_assert(S,G,rpt) )

inherited_olist(S,G) =
    inherited_olist(S,G,rpt) (+)
    joins(S,G) (+) pim_include(S,G) (-) lost_assert(S,G)
```

The macros pim_include(*,G) and pim_include(S,G) indicate the
interfaces to which traffic might be forwarded because of hosts that
are local members on that interface. Note that normally only the DR
cares about local membership, but when an assert happens, the assert
winner takes over responsibility for forwarding traffic to local
members that have requested traffic on a group or source/group pair.

```
pim_include(*,G) =
    { all interfaces I such that:
      ( ( I_am_DR( I ) AND lost_assert(*,G,I) == FALSE )
        OR AssertWinner(*,G,I) == me )
      AND  local_receiver_include(*,G,I) }

pim_include(S,G) =
    { all interfaces I such that:
      ( (I_am_DR( I ) AND lost_assert(S,G,I) == FALSE )
        OR AssertWinner(S,G,I) == me )
        AND  local_receiver_include(S,G,I) }

pim_exclude(S,G) =
    { all interfaces I such that:
      ( (I_am_DR( I ) AND lost_assert(*,G,I) == FALSE )
        OR AssertWinner(*,G,I) == me )
        AND  local_receiver_exclude(S,G,I) }
```

The clause "local_receiver_include(S,G,I)" is true if the IGMP/MLD
module or other local membership mechanism has determined that local
members on interface I desire to receive traffic sent specifically by
S to G. "local_receiver_include(*,G,I)" is true if the IGMP/MLD
module or other local membership mechanism has determined that local

members on interface I desire to receive all traffic sent to G
(possibly excluding traffic from a specific set of sources).
"local_receiver_exclude(S,G,I) is true if
"local_receiver_include(*,G,I)" is true but none of the local members
desire to receive traffic from S.

The set "joins(*,*,RP)" is the set of all interfaces on which the
router has received (*,*,RP) Joins:

    joins(*,*,RP) =
        { all interfaces I such that
          DownstreamJPState(*,*,RP,I) is either Join or
              Prune-Pending }

DownstreamJPState(*,*,RP,I) is the state of the finite state machine
in Section 4.5.1.

The set "joins(*,G)" is the set of all interfaces on which the router
has received (*,G) Joins:

    joins(*,G) =
        { all interfaces I such that
          DownstreamJPState(*,G,I) is either Join or Prune-Pending }

DownstreamJPState(*,G,I) is the state of the finite state machine in
Section 4.5.2.

The set "joins(S,G)" is the set of all interfaces on which the router
has received (S,G) Joins:

    joins(S,G) =
        { all interfaces I such that
          DownstreamJPState(S,G,I) is either Join or Prune-Pending }

DownstreamJPState(S,G,I) is the state of the finite state machine in
Section 4.5.3.

The set "prunes(S,G,rpt)" is the set of all interfaces on which the
router has received (*,G) joins and (S,G,rpt) prunes.

    prunes(S,G,rpt) =
        { all interfaces I such that
          DownstreamJPState(S,G,rpt,I) is Prune or PruneTmp }

DownstreamJPState(S,G,rpt,I) is the state of the finite state machine
in Section 4.5.4.

The set "lost_assert(*,G)" is the set of all interfaces on which the
router has received (*,G) joins but has lost a (*,G) assert. The
macro lost_assert(*,G,I) is defined in Section 4.6.5.

```
lost_assert(*,G) =
    { all interfaces I such that
      lost_assert(*,G,I) == TRUE }
```

The set "lost_assert(S,G,rpt)" is the set of all interfaces on which
the router has received (*,G) joins but has lost an (S,G) assert.
The macro lost_assert(S,G,rpt,I) is defined in Section 4.6.5.

```
lost_assert(S,G,rpt) =
    { all interfaces I such that
      lost_assert(S,G,rpt,I) == TRUE }
```

The set "lost_assert(S,G)" is the set of all interfaces on which the
router has received (S,G) joins but has lost an (S,G) assert. The
macro lost_assert(S,G,I) is defined in Section 4.6.5.

```
lost_assert(S,G) =
    { all interfaces I such that
      lost_assert(S,G,I) == TRUE }
```

The following pseudocode macro definitions are also used in many
places in the specification. Basically, RPF' is the RPF neighbor
towards an RP or source unless a PIM-Assert has overridden the normal
choice of neighbor.

```
  neighbor RPF'(*,G) {
      if ( I_Am_Assert_Loser(*, G, RPF_interface(RP(G))) ) {
          return AssertWinner(*, G, RPF_interface(RP(G)) )
      } else {
          return NBR( RPF_interface(RP(G)), MRIB.next_hop( RP(G) ) )
      }
  }

  neighbor RPF'(S,G,rpt) {
      if( I_Am_Assert_Loser(S, G, RPF_interface(RP(G)) ) ) {
          return AssertWinner(S, G, RPF_interface(RP(G)) )
      } else {
          return RPF'(*,G)
      }
  }
```

```
   neighbor RPF'(S,G) {
       if ( I_Am_Assert_Loser(S, G, RPF_interface(S) )) {
           return AssertWinner(S, G, RPF_interface(S) )
       } else {
           return NBR( RPF_interface(S), MRIB.next_hop( S ) )
       }
   }
```

RPF'(*,G) and RPF'(S,G) indicate the neighbor from which data packets
should be coming and to which joins should be sent on the RP tree and
SPT, respectively.

RPF'(S,G,rpt) is basically RPF'(*,G) modified by the result of an
Assert(S,G) on RPF_interface(RP(G)). In such a case, packets from S
will be originating from a different router than RPF'(*,G). If we
only have active (*,G) Join state, we need to accept packets from
RPF'(S,G,rpt) and add a Prune(S,G,rpt) to the periodic Join(*,G)
messages that we send to RPF'(*,G) (see Section 4.5.8).

The function MRIB.next_hop(S) returns an address of the next-hop
PIM neighbor toward the host S, as indicated by the current MRIB. If
S is directly adjacent, then MRIB.next_hop(S) returns NULL. At the
RP for G, MRIB.next_hop(RP(G)) returns NULL.

The function NBR(I, A) uses information gathered through PIM Hello
messages to map the IP address A of a directly connected PIM neighbor
router on interface I to the primary IP address of the same router
(Section 4.3.4). The primary IP address of a neighbor is the address
that it uses as the source of its PIM Hello messages. Note that a
neighbor's IP address may be non-unique within the PIM neighbor
database due to scope issues. The address must, however, be unique
amongst the addresses of all the PIM neighbors on a specific
interface.

I_Am_Assert_Loser(S, G, I) is true if the Assert state machine (in
Section 4.6.1) for (S,G) on Interface I is in "I am Assert Loser"
state.

I_Am_Assert_Loser(*, G, I) is true if the Assert state machine (in
Section 4.6.2) for (*,G) on Interface I is in "I am Assert Loser"
state.

4.2. Data Packet Forwarding Rules

   The PIM-SM packet forwarding rules are defined below in pseudocode.

      iif is the incoming interface of the packet.
      S is the source address of the packet.
      G is the destination address of the packet (group address).
      RP is the address of the Rendezvous Point for this group.
      RPF_interface(S) is the interface the MRIB indicates would be used
      to route packets to S.
      RPF_interface(RP) is the interface the MRIB indicates would be
      used to route packets to RP, except at the RP when it is the
      decapsulation interface (the "virtual" interface on which register
      packets are received).

   First, we restart (or start) the Keepalive Timer if the source is on
   a directly connected subnet.

   Second, we check to see if the SPTbit should be set because we've now
   switched from the RP tree to the SPT.

   Next, we check to see whether the packet should be accepted based on
   TIB state and the interface that the packet arrived on.

   If the packet should be forwarded using (S,G) state, we then build an
   outgoing interface list for the packet. If this list is not empty,
   then we restart the (S,G) state Keepalive Timer.

   If the packet should be forwarded using (*,*,RP) or (*,G) state, then
   we just build an outgoing interface list for the packet. We also
   check if we should initiate a switch to start receiving this source
   on a shortest path tree.

   Finally we remove the incoming interface from the outgoing interface
   list we've created, and if the resulting outgoing interface list is
   not empty, we forward the packet out of those interfaces.

```
On receipt of data from S to G on interface iif:
 if( DirectlyConnected(S) == TRUE AND iif == RPF_interface(S) ) {
      set KeepaliveTimer(S,G) to Keepalive_Period
       # Note: a register state transition or UpstreamJPState(S,G)
       # transition may happen as a result of restarting
       # KeepaliveTimer, and must be dealt with here.
 }

 if( iif == RPF_interface(S) AND UpstreamJPState(S,G) == Joined AND
    inherited_olist(S,G) != NULL ) {
       set KeepaliveTimer(S,G) to Keepalive_Period
 }

 Update_SPTbit(S,G,iif)
 oiflist = NULL

 if( iif == RPF_interface(S) AND SPTbit(S,G) == TRUE ) {
   oiflist = inherited_olist(S,G)
 } else if( iif == RPF_interface(RP(G)) AND SPTbit(S,G) == FALSE) {
   oiflist = inherited_olist(S,G,rpt)
   CheckSwitchToSpt(S,G)
 } else {
   # Note: RPF check failed
   # A transition in an Assert FSM may cause an Assert(S,G)
   # or Assert(*,G) message to be sent out interface iif.
   # See section 4.6 for details.
   if ( SPTbit(S,G) == TRUE AND iif is in inherited_olist(S,G) ) {
      send Assert(S,G) on iif
   } else if ( SPTbit(S,G) == FALSE AND
               iif is in inherited_olist(S,G,rpt) {
      send Assert(*,G) on iif
   }
 }

 oiflist = oiflist (-) iif
 forward packet on all interfaces in oiflist
```

This pseudocode employs several "macro" definitions:

DirectlyConnected(S) is TRUE if the source S is on any subnet that is directly connected to this router (or for packets originating on this router).

inherited_olist(S,G) and inherited_olist(S,G,rpt) are defined in Section 4.1.

Basically, inherited_olist(S,G) is the outgoing interface list for
packets forwarded on (S,G) state, taking into account (*,*,RP) state,
(*,G) state, asserts, etc.

inherited_olist(S,G,rpt) is the outgoing interface list for packets
forwarded on (*,*,RP) or (*,G) state, taking into account (S,G,rpt)
prune state, asserts, etc.

Update_SPTbit(S,G,iif) is defined in Section 4.2.2.

CheckSwitchToSpt(S,G) is defined in Section 4.2.1.

UpstreamJPState(S,G) is the state of the finite state machine in
Section 4.5.7.

Keepalive_Period is defined in Section 4.10.

Data-triggered PIM-Assert messages sent from the above forwarding
code should be rate-limited in a implementation-dependent manner.

4.2.1. Last-Hop Switchover to the SPT

In Sparse-Mode PIM, last-hop routers join the shared tree towards the
RP. Once traffic from sources to joined groups arrives at a last-hop
router, it has the option of switching to receive the traffic on a
shortest path tree (SPT).

The decision for a router to switch to the SPT is controlled as
follows:

```
  void
  CheckSwitchToSpt(S,G) {
    if ( ( pim_include(*,G) (-) pim_exclude(S,G)
          (+) pim_include(S,G) != NULL )
        AND SwitchToSptDesired(S,G) ) {
           # Note: Restarting the KAT will result in the SPT switch
           set KeepaliveTimer(S,G) to Keepalive_Period
    }
  }
```

SwitchToSptDesired(S,G) is a policy function that is implementation
defined. An "infinite threshold" policy can be implemented by making
SwitchToSptDesired(S,G) return false all the time. A "switch on
first packet" policy can be implemented by making
SwitchToSptDesired(S,G) return true once a single packet has been
received for the source and group.

4.2.2. Setting and Clearing the (S,G) SPTbit

   The (S,G) SPTbit is used to distinguish whether to forward on
   (*,*,RP)/(*,G) or on (S,G) state. When switching from the RP tree to
   the source tree, there is a transition period when data is arriving
   due to upstream (*,*,RP)/(*,G) state while upstream (S,G) state is
   being established, during which time a router should continue to
   forward only on (*,*,RP)/(*,G) state. This prevents temporary
   black-holes that would be caused by sending a Prune(S,G,rpt) before
   the upstream (S,G) state has finished being established.

   Thus, when a packet arrives, the (S,G) SPTbit is updated as follows:

```
     void
     Update_SPTbit(S,G,iif) {
       if ( iif == RPF_interface(S)
             AND JoinDesired(S,G) == TRUE
             AND ( DirectlyConnected(S) == TRUE
                   OR RPF_interface(S) != RPF_interface(RP(G))
                   OR inherited_olist(S,G,rpt) == NULL
                   OR ( ( RPF'(S,G) == RPF'(*,G) ) AND
                        ( RPF'(S,G) != NULL ) )
                   OR ( I_Am_Assert_Loser(S,G,iif) ) {
          Set SPTbit(S,G) to TRUE
       }
     }
```

   Additionally, a router can set SPTbit(S,G) to TRUE in other cases,
   such as when it receives an Assert(S,G) on RPF_interface(S) (see
   Section 4.6.1).

   JoinDesired(S,G) is defined in Section 4.5.7 and indicates whether we
   have the appropriate (S,G) Join state to wish to send a Join(S,G)
   upstream.

   Basically, Update_SPTbit will set the SPTbit if we have the
   appropriate (S,G) join state, and if the packet arrived on the
   correct upstream interface for S, and if one or more of the following
   conditions applies:

   1. The source is directly connected, in which case the switch to the
       SPT is a no-op.

   2. The RPF interface to S is different from the RPF interface to the
       RP. The packet arrived on RPF_interface(S), and so the SPT must
       have been completed.

   3. Noone wants the packet on the RP tree.

    4. RPF'(S,G) == RPF'(*,G). In this case, the router will never be
        able to tell if the SPT has been completed, so it should just
        switch immediately.

    In the case where the RPF interface is the same for the RP and for S,
    but RPF'(S,G) and RPF'(*,G) differ, we wait for an Assert(S,G), which
    indicates that the upstream router with (S,G) state believes the SPT
    has been completed. However, item (3) above is needed because there
    may not be any (*,G) state to trigger an Assert(S,G) to happen.

    The SPTbit is cleared in the (S,G) upstream state machine (see
    Section 4.5.7) when JoinDesired(S,G) becomes FALSE.

4.3. Designated Routers (DR) and Hello Messages

    A shared-media LAN like Ethernet may have multiple PIM-SM routers
    connected to it. A single one of these routers, the DR, will act on
    behalf of directly connected hosts with respect to the PIM-SM
    protocol. Because the distinction between LANs and point-to-point
    interfaces can sometimes be blurred, and because routers may also
    have multicast host functionality, the PIM-SM specification makes no
    distinction between the two. Thus, DR election will happen on all
    interfaces, LAN or otherwise.

    DR election is performed using Hello messages. Hello messages are
    also the way that option negotiation takes place in PIM, so that
    additional functionality can be enabled, or parameters tuned.

4.3.1. Sending Hello Messages

    PIM Hello messages are sent periodically on each PIM-enabled
    interface. They allow a router to learn about the neighboring PIM
    routers on each interface. Hello messages are also the mechanism
    used to elect a Designated Router (DR), and to negotiate additional
    capabilities. A router must record the Hello information received
    from each PIM neighbor.

    Hello messages MUST be sent on all active interfaces, including
    physical point-to-point links, and are multicast to the 'ALL-PIM-
    ROUTERS' group address ('224.0.0.13' for IPv4 and 'ff02::d' for
    IPv6).

      We note that some implementations do not send Hello messages on
      point-to-point interfaces. This is non-compliant behavior. A
      compliant PIM router MUST send Hello messages, even on point-to-
      point interfaces.

A per-interface Hello Timer (HT(I)) is used to trigger sending Hello
messages on each active interface. When PIM is enabled on an
interface or a router first starts, the Hello Timer of that interface
is set to a random value between 0 and Triggered_Hello_Delay. This
prevents synchronization of Hello messages if multiple routers are
powered on simultaneously. After the initial randomized interval,
Hello messages must be sent every Hello_Period seconds. The Hello
Timer should not be reset except when it expires.

Note that neighbors will not accept Join/Prune or Assert messages
from a router unless they have first heard a Hello message from that
router. Thus, if a router needs to send a Join/Prune or Assert
message on an interface on which it has not yet sent a Hello message
with the currently configured IP address, then it MUST immediately
send the relevant Hello message without waiting for the Hello Timer
to expire, followed by the Join/Prune or Assert message.

The DR_Priority Option allows a network administrator to give
preference to a particular router in the DR election process by
giving it a numerically larger DR Priority. The DR_Priority Option
SHOULD be included in every Hello message, even if no DR Priority is
explicitly configured on that interface. This is necessary because
priority-based DR election is only enabled when all neighbors on an
interface advertise that they are capable of using the DR_Priority
Option. The default priority is 1.

The Generation_Identifier (GenID) Option SHOULD be included in all
Hello messages. The GenID option contains a randomly generated
32-bit value that is regenerated each time PIM forwarding is started
or restarted on the interface, including when the router itself
restarts. When a Hello message with a new GenID is received from a
neighbor, any old Hello information about that neighbor SHOULD be
discarded and superseded by the information from the new Hello
message. This may cause a new DR to be chosen on that interface.

The LAN Prune Delay Option SHOULD be included in all Hello messages
sent on multi-access LANs. This option advertises a router's
capability to use values other than the defaults for the
Propagation_Delay and Override_Interval, which affect the setting of
the Prune-Pending, Upstream Join, and Override Timers (defined in
Section 4.10).

The Address List Option advertises all the secondary addresses
associated with the source interface of the router originating the
message. The option MUST be included in all Hello messages if there
are secondary addresses associated with the source interface and MAY
be omitted if no secondary addresses exist.

To allow new or rebooting routers to learn of PIM neighbors quickly,
when a Hello message is received from a new neighbor, or a Hello
message with a new GenID is received from an existing neighbor, a new
Hello message should be sent on this interface after a randomized
delay between 0 and Triggered_Hello_Delay. This triggered message
need not change the timing of the scheduled periodic message. If a
router needs to send a Join/Prune to the new neighbor or send an
Assert message in response to an Assert message from the new neighbor
before this randomized delay has expired, then it MUST immediately
send the relevant Hello message without waiting for the Hello Timer
to expire, followed by the Join/Prune or Assert message. If it does
not do this, then the new neighbor will discard the Join/Prune or
Assert message.

Before an interface goes down or changes primary IP address, a Hello
message with a zero HoldTime should be sent immediately (with the old
IP address if the IP address changed). This will cause PIM neighbors
to remove this neighbor (or its old IP address) immediately. After
an interface has changed its IP address, it MUST send a Hello message
with its new IP address. If an interface changes one of its
secondary IP addresses, a Hello message with an updated Address_List
option and a non-zero HoldTime should be sent immediately. This will
cause PIM neighbors to update this neighbor's list of secondary
addresses immediately.

4.3.2. DR Election

When a PIM Hello message is received on interface I, the following
information about the sending neighbor is recorded:

   neighbor.interface
        The interface on which the Hello message arrived.

   neighbor.primary_ip_address
        The IP address that the PIM neighbor used as the source
        address of the Hello message.

   neighbor.genid
        The Generation ID of the PIM neighbor.

   neighbor.dr_priority
        The DR Priority field of the PIM neighbor, if it is present in
        the Hello message.

   neighbor.dr_priority_present
        A flag indicating if the DR Priority field was present in the
        Hello message.

neighbor.timeout
     A timer value to time out the neighbor state when it becomes
     stale, also known as the Neighbor Liveness Timer.

     The Neighbor Liveness Timer (NLT(N,I)) is reset to
     Hello_Holdtime (from the Hello Holdtime option) whenever a
     Hello message is received containing a Holdtime option, or to
     Default_Hello_Holdtime if the Hello message does not contain
     the Holdtime option.

     Neighbor state is deleted when the neighbor timeout expires.

The function for computing the DR on interface I is:

```
  host
  DR(I) {
       dr = me
       for each neighbor on interface I {
           if ( dr_is_better( neighbor, dr, I ) == TRUE ) {
                dr = neighbor
           }
       }
       return dr
  }
```

The function used for comparing DR "metrics" on interface I is:

```
  bool
  dr_is_better(a,b,I) {
       if( there is a neighbor n on I for which n.dr_priority_present
               is false ) {
           return a.primary_ip_address > b.primary_ip_address
       } else {
           return ( a.dr_priority > b.dr_priority ) OR
                  ( a.dr_priority == b.dr_priority AND
                     a.primary_ip_address > b.primary_ip_address )
       }
  }
```

The trivial function I_am_DR(I) is defined to aid readability:

```
  bool
  I_am_DR(I) {
     return DR(I) == me
  }
```

The DR Priority is a 32-bit unsigned number, and the numerically
larger priority is always preferred. A router's idea of the current
DR on an interface can change when a PIM Hello message is received,
when a neighbor times out, or when a router's own DR Priority
changes. If the router becomes the DR or ceases to be the DR, this
will normally cause the DR Register state machine to change state.
Subsequent actions are determined by that state machine.

   We note that some PIM implementations do not send Hello messages on
   point-to-point interfaces and thus cannot perform DR election on
   such interfaces. This is non-compliant behavior. DR election MUST
   be performed on ALL active PIM-SM interfaces.

4.3.3. Reducing Prune Propagation Delay on LANs

   In addition to the information recorded for the DR Election, the
   following per neighbor information is obtained from the LAN Prune
   Delay Hello option:

   neighbor.lan_prune_delay_present
        A flag indicating if the LAN Prune Delay option was present in
        the Hello message.

   neighbor.tracking_support
        A flag storing the value of the T bit in the LAN Prune Delay
        option if it is present in the Hello message. This indicates
        the neighbor's capability to disable Join message suppression.

   neighbor.propagation_delay
        The Propagation Delay field of the LAN Prune Delay option (if
        present) in the Hello message.

   neighbor.override_interval
        The Override_Interval field of the LAN Prune Delay option (if
        present) in the Hello message.

   The additional state described above is deleted along with the DR
   neighbor state when the neighbor timeout expires.

   Just like the DR_Priority option, the information provided in the LAN
   Prune Delay option is not used unless all neighbors on a link
   advertise the option. The function below computes this state:

```
    bool
    lan_delay_enabled(I) {
        for each neighbor on interface I {
            if ( neighbor.lan_prune_delay_present == false ) {
                return false
            }
        }
        return true
    }
```

The Propagation Delay inserted by a router in the LAN Prune Delay
option expresses the expected message propagation delay on the link
and should be configurable by the system administrator. It is used
by upstream routers to figure out how long they should wait for a
Join override message before pruning an interface.

PIM implementers should enforce a lower bound on the permitted values
for this delay to allow for scheduling and processing delays within
their router. Such delays may cause received messages to be
processed later as well as triggered messages to be sent later than
intended. Setting this Propagation Delay to too low a value may
result in temporary forwarding outages because a downstream router
will not be able to override a neighbor's Prune message before the
upstream neighbor stops forwarding.

When all routers on a link are in a position to negotiate a
Propagation Delay different from the default, the largest value from
those advertised by each neighbor is chosen. The function for
computing the Effective_Propagation_Delay of interface I is:

```
  time_interval
  Effective_Propagation_Delay(I) {
      if ( lan_delay_enabled(I) == false ) {
          return Propagation_delay_default
      }
      delay = Propagation_Delay(I)
      for each neighbor on interface I {
          if ( neighbor.propagation_delay > delay ) {
              delay = neighbor.propagation_delay
          }
      }
      return delay
  }
```

To avoid synchronization of override messages when multiple
downstream routers share a multi-access link, sending of such
messages is delayed by a small random amount of time. The period of
randomization should represent the size of the PIM router population

on the link. Each router expresses its view of the amount of
randomization necessary in the Override Interval field of the LAN
Prune Delay option.

When all routers on a link are in a position to negotiate an Override
Interval different from the default, the largest value from those
advertised by each neighbor is chosen. The function for computing
the Effective Override Interval of interface I is:

```
  time_interval
  Effective_Override_Interval(I) {
      if ( lan_delay_enabled(I) == false ) {
          return t_override_default
      }
      delay = Override_Interval(I)
      for each neighbor on interface I {
          if ( neighbor.override_interval > delay ) {
              delay = neighbor.override_interval
          }
      }
      return delay
  }
```

Although the mechanisms are not specified in this document, it is
possible for upstream routers to explicitly track the join membership
of individual downstream routers if Join suppression is disabled. A
router can advertise its willingness to disable Join suppression by
using the T bit in the LAN Prune Delay Hello option. Unless all PIM
routers on a link negotiate this capability, explicit tracking and
the disabling of the Join suppression mechanism are not possible.
The function for computing the state of Suppression on interface I
is:

```
  bool
  Suppression_Enabled(I) {
      if ( lan_delay_enabled(I) == false ) {
          return true
      }
      for each neighbor on interface I {
          if ( neighbor.tracking_support == false ) {
              return true
          }
      }
      return false
  }
```

Note that the setting of Suppression_Enabled(I) affects the value of
t_suppressed (see Section 4.10).

4.3.4. Maintaining Secondary Address Lists

   Communication of a router's interface secondary addresses to its PIM
   neighbors is necessary to provide the neighbors with a mechanism for
   mapping next_hop information obtained through their MRIB to a primary
   address that can be used as a destination for Join/Prune messages.
   The mapping is performed through the NBR macro. The primary address
   of a PIM neighbor is obtained from the source IP address used in its
   PIM Hello messages. Secondary addresses are carried within the Hello
   message in an Address List Hello option. The primary address of the
   source interface of the router MUST NOT be listed within the Address
   List Hello option.

   In addition to the information recorded for the DR Election, the
   following per neighbor information is obtained from the Address List
   Hello option:

     neighbor.secondary_address_list
           The list of secondary addresses used by the PIM neighbor on
           the interface through which the Hello message was transmitted.

   When processing a received PIM Hello message containing an Address
   List Hello option, the list of secondary addresses in the message
   completely replaces any previously associated secondary addresses for
   that neighbor. If a received PIM Hello message does not contain an
   Address List Hello option, then all secondary addresses associated
   with the neighbor must be deleted. If a received PIM Hello message
   contains an Address List Hello option that includes the primary
   address of the sending router in the list of secondary addresses
   (although this is not expected), then the addresses listed in the
   message, excluding the primary address, are used to update the
   associated secondary addresses for that neighbor.

   All the advertised secondary addresses in received Hello messages
   must be checked against those previously advertised by all other PIM
   neighbors on that interface. If there is a conflict and the same
   secondary address was previously advertised by another neighbor, then
   only the most recently received mapping MUST be maintained, and an
   error message SHOULD be logged to the administrator in a rate-limited
   manner.

   Within one Address List Hello option, all the addresses MUST be of
   the same address family. It is not permitted to mix IPv4 and IPv6
   addresses within the same message. In addition, the address family
   of the fields in the message SHOULD be the same as the IP source and
   destination addresses of the packet header.

4.4. PIM Register Messages

   The Designated Router (DR) on a LAN or point-to-point link
   encapsulates multicast packets from local sources to the RP for the
   relevant group unless it recently received a Register-Stop message
   for that (S,G) or (*,G) from the RP. When the DR receives a
   Register-Stop message from the RP, it starts a Register-Stop Timer to
   maintain this state. Just before the Register-Stop Timer expires,
   the DR sends a Null-Register Message to the RP to allow the RP to
   refresh the Register-Stop information at the DR. If the Register-
   Stop Timer actually expires, the DR will resume encapsulating packets
   from the source to the RP.

4.4.1. Sending Register Messages from the DR

   Every PIM-SM router has the capability to be a DR. The state machine
   below is used to implement Register functionality. For the purposes
   of specification, we represent the mechanism to encapsulate packets
   to the RP as a Register-Tunnel interface, which is added to or
   removed from the (S,G) olist. The tunnel interface then takes part
   in the normal packet forwarding rules as specified in Section 4.2.

   If register state is maintained, it is maintained only for directly
   connected sources and is per-(S,G). There are four states in the
   DR's per-(S,G) Register state machine:

   Join (J)
        The register tunnel is "joined" (the join is actually implicit,
        but the DR acts as if the RP has joined the DR on the tunnel
        interface).

   Prune (P)
        The register tunnel is "pruned" (this occurs when a Register-
        Stop is received).

   Join-Pending (JP)
        The register tunnel is pruned but the DR is contemplating adding
        it back.

   NoInfo (NI)
        No information. This is the initial state, and the state when
        the router is not the DR.

   In addition, a Register-Stop Timer (RST) is kept if the state machine
   is not in the NoInfo state.

Figure 1: Per-(S,G) register state machine at a DR in tabular form

		Event			
Prev State	Register-Stop Timer expires	Could Register ->True	Could Register ->False	Register-Stop received	RP changed
NoInfo (NI)	-	-> J state add reg tunnel	-	-	-
Join (J)	-	-	-> NI state remove reg tunnel	-> P state remove reg tunnel; set Register-Stop Timer(*)	-> J state update reg tunnel
Join-Pending (JP)	-> J state add reg tunnel	-	-> NI state	-> P state set Register-Stop Timer(*)	-> J state add reg tunnel; cancel Register-Stop Timer
Prune (P)	-> JP state set Register-Stop Timer(**); send Null-Register	-	-> NI state	-	-> J state add reg tunnel; cancel Register-Stop Timer

   Notes:

   (*) The Register-Stop Timer is set to a random value chosen
        uniformly from the interval (0.5 * Register_Suppression_Time,
        1.5 * Register_Suppression_Time) minus Register_Probe_Time.

Subtracting off Register_Probe_Time is a bit unnecessary because
it is really small compared to Register_Suppression_Time, but
this was in the old spec and is kept for compatibility.

(**) The Register-Stop Timer is set to Register_Probe_Time.

The following three actions are defined:

Add Register Tunnel

A Register-Tunnel virtual interface, VI, is created (if it doesn't
already exist) with its encapsulation target being RP(G).
DownstreamJPState(S,G,VI) is set to Join state, causing the tunnel
interface to be added to immediate_olist(S,G) and
inherited_olist(S,G).

Remove Register Tunnel

VI is the Register-Tunnel virtual interface with encapsulation
target of RP(G). DownstreamJPState(S,G,VI) is set to NoInfo
state, causing the tunnel interface to be removed from
immediate_olist(S,G) and inherited_olist(S,G). If
DownstreamJPState(S,G,VI) is NoInfo for all (S,G), then VI can be
deleted.

Update Register Tunnel

This action occurs when RP(G) changes.

VI_old is the Register-Tunnel virtual interface with encapsulation
target old_RP(G). A Register-Tunnel virtual interface, VI_new, is
created (if it doesn't already exist) with its encapsulation
target being new_RP(G). DownstreamJPState(S,G,VI_old) is set to
NoInfo state and DownstreamJPState(S,G,VI_new) is set to Join
state. If DownstreamJPState(S,G,VI_old) is NoInfo for all (S,G),
then VI_old can be deleted.

Note that we cannot simply change the encapsulation target of
VI_old because not all groups using that encapsulation tunnel will
have moved to the same new RP.

CouldRegister(S,G)

 The macro "CouldRegister" in the state machine is defined as:

```
bool CouldRegister(S,G) {
   return ( I_am_DR( RPF_interface(S) ) AND
            KeepaliveTimer(S,G) is running AND
            DirectlyConnected(S) == TRUE )
}
```

 Note that on reception of a packet at the DR from a directly
 connected source, KeepaliveTimer(S,G) needs to be set by the
 packet forwarding rules before computing CouldRegister(S,G) in the
 register state machine, or the first packet from a source won't be
 registered.

Encapsulating Data Packets in the Register Tunnel

 Conceptually, the Register Tunnel is an interface with a smaller
 MTU than the underlying IP interface towards the RP. IP
 fragmentation on packets forwarded on the Register Tunnel is
 performed based upon this smaller MTU. The encapsulating DR may
 perform Path MTU Discovery to the RP to determine the effective
 MTU of the tunnel. Fragmentation for the smaller MTU should take
 both the outer IP header and the PIM register header overhead into
 account. If a multicast packet is fragmented on the way into the
 Register Tunnel, each fragment is encapsulated individually so it
 contains IP, PIM, and inner IP headers.

 In IPv6, the DR MUST perform Path MTU discovery, and an ICMP
 Packet Too Big message MUST be sent by the encapsulating DR if it
 receives a packet that will not fit in the effective MTU of the
 tunnel. If the MTU between the DR and the RP results in the
 effective tunnel MTU being smaller than 1280 (the IPv6 minimum
 MTU), the DR MUST send Fragmentation Required messages with an MTU
 value of 1280 and MUST fragment its PIM register messages as
 required, using an IPv6 fragmentation header between the outer
 IPv6 header and the PIM Register header.

 The TTL of a forwarded data packet is decremented before it is
 encapsulated in the Register Tunnel. The encapsulating packet
 uses the normal TTL that the router would use for any locally-
 generated IP packet.

 The IP ECN bits should be copied from the original packet to the
 IP header of the encapsulating packet. They SHOULD NOT be set
 independently by the encapsulating router.

The Diffserv Code Point (DSCP) should be copied from the original
packet to the IP header of the encapsulating packet. It MAY be
set independently by the encapsulating router, based upon static
configuration or traffic classification. See [12] for more
discussion on setting the DSCP on tunnels.

Handling Register-Stop(*,G) Messages at the DR

An old RP might send a Register-Stop message with the source
address set to all zeros. This was the normal course of action in
RFC 2362 when the Register message matched against (*,G) state at
the RP, and it was defined as meaning "stop encapsulating all
sources for this group". However, the behavior of such a
Register-Stop(*,G) is ambiguous or incorrect in some
circumstances.

We specify that an RP should not send Register-Stop(*,G) messages,
but for compatibility, a DR should be able to accept one if it is
received.

A Register-Stop(*,G) should be treated as a Register-Stop(S,G) for
all (S,G) Register state machines that are not in the NoInfo
state. A router should not apply a Register-Stop(*,G) to sources
that become active after the Register-Stop(*,G) was received.

4.4.2. Receiving Register Messages at the RP

   When an RP receives a Register message, the course of action is
   decided according to the following pseudocode:

```
packet_arrives_on_rp_tunnel( pkt ) {
    if( outer.dst is not one of my addresses ) {
        drop the packet silently.
        # Note: this may be a spoofing attempt
    }
    if( I_am_RP(G) AND outer.dst == RP(G) ) {
          sentRegisterStop = FALSE;
          if ( register.borderbit == TRUE ) {
               if ( PMBR(S,G) == unknown ) {
                   PMBR(S,G) = outer.src
               } else if ( outer.src != PMBR(S,G) ) {
                   send Register-Stop(S,G) to outer.src
                   drop the packet silently.
               }
          }
          if ( SPTbit(S,G) OR
           ( SwitchToSptDesired(S,G) AND
             ( inherited_olist(S,G) == NULL ))) {
            send Register-Stop(S,G) to outer.src
            sentRegisterStop = TRUE;
          }
          if ( SPTbit(S,G) OR SwitchToSptDesired(S,G) ) {
               if ( sentRegisterStop == TRUE ) {
                   set KeepaliveTimer(S,G) to RP_Keepalive_Period;
               } else {
                   set KeepaliveTimer(S,G) to Keepalive_Period;
               }
          }
          if( !SPTbit(S,G) AND ! pkt.NullRegisterBit ) {
               decapsulate and forward the inner packet to
               inherited_olist(S,G,rpt) # Note (+)
          }
    } else {
        send Register-Stop(S,G) to outer.src
        # Note (*)
    }
}
```

   outer.dst is the IP destination address of the encapsulating header.

   outer.src is the IP source address of the encapsulating header, i.e.,
   the DR's address.

I_am_RP(G) is true if the group-to-RP mapping indicates that this
router is the RP for the group.

Note (*): This may block traffic from S for Register_Suppression_Time
   if the DR learned about a new group-to-RP mapping before the RP
   did. However, this doesn't matter unless we figure out some way
   for the RP also to accept (*,G) joins when it doesn't yet realize
   that it is about to become the RP for G. This will all get sorted
   out once the RP learns the new group-to-rp mapping. We decided to
   do nothing about this and just accept the fact that PIM may suffer
   interrupted (*,G) connectivity following an RP change.

Note (+): Implementations are advised not to make this a special
   case, but to arrange that this path rejoin the normal packet
   forwarding path. All of the appropriate actions from the "On
   receipt of data from S to G on interface iif" pseudocode in
   Section 4.2 should be performed.

KeepaliveTimer(S,G) is restarted at the RP when packets arrive on the
proper tunnel interface and the RP desires to switch to the SPT or
the SPTbit is already set. This may cause the upstream (S,G) state
machine to trigger a join if the inherited_olist(S,G) is not NULL.

An RP should preserve (S,G) state that was created in response to a
Register message for at least (3 * Register_Suppression_Time);
otherwise, the RP may stop joining (S,G) before the DR for S has
restarted sending registers. Traffic would then be interrupted until
the Register-Stop Timer expires at the DR.

Thus, at the RP, KeepaliveTimer(S,G) should be restarted to (3 *
Register_Suppression_Time + Register_Probe_Time).

When forwarding a packet from the Register Tunnel, the TTL of the
original data packet is decremented after it is decapsulated.

The IP ECN bits should be copied from the IP header of the Register
packet to the decapsulated packet.

The Diffserv Code Point (DSCP) should be copied from the IP header of
the Register packet to the decapsulated packet. The RP MAY retain
the DSCP of the inner packet or re-classify the packet and apply a
different DSCP. Scenarios where each of these might be useful are
discussed in [12].

4.5. PIM Join/Prune Messages

   A PIM Join/Prune message consists of a list of groups and a list of
   Joined and Pruned sources for each group. When processing a received
   Join/Prune message, each Joined or Pruned source for a Group is
   effectively considered individually, and applies to one or more of
   the following state machines. When considering a Join/Prune message
   whose Upstream Neighbor Address field addresses this router, (*,G)
   Joins and Prunes can affect both the (*,G) and (S,G,rpt) downstream
   state machines, while (*,*,RP), (S,G), and (S,G,rpt) Joins and Prunes
   can only affect their respective downstream state machines. When
   considering a Join/Prune message whose Upstream Neighbor Address
   field addresses another router, most Join or Prune messages could
   affect each upstream state machine.

   In general, a PIM Join/Prune message should only be accepted for
   processing if it comes from a known PIM neighbor. A PIM router hears
   about PIM neighbors through PIM Hello messages. If a router receives
   a Join/Prune message from a particular IP source address and it has
   not seen a PIM Hello message from that source address, then the
   Join/Prune message SHOULD be discarded without further processing.
   In addition, if the Hello message from a neighbor was authenticated
   using IPsec AH (see Section 6.3), then all Join/Prune messages from
   that neighbor MUST also be authenticated using IPsec AH.

   We note that some older PIM implementations incorrectly fail to send
   Hello messages on point-to-point interfaces, so we also RECOMMEND
   that a configuration option be provided to allow interoperation with
   such older routers, but that this configuration option SHOULD NOT be
   enabled by default.

4.5.1. Receiving (*,*,RP) Join/Prune Messages

   The per-interface state machine for receiving (*,*,RP) Join/Prune
   Messages is given below. There are three states:

     NoInfo (NI)
          The interface has no (*,*,RP) Join state and no timers
          running.

     Join (J)
          The interface has (*,*,RP) Join state, which will cause the
          router to forward packets destined for any group handled by RP
          from this interface except if there is also (S,G,rpt) prune
          information (see Section 4.5.4) or the router lost an assert
          on this interface.

Prune-Pending (PP)
      The router has received a Prune(*,*,RP) on this interface from
      a downstream neighbor and is waiting to see whether the prune
      will be overridden by another downstream router. For
      forwarding purposes, the Prune-Pending state functions exactly
      like the Join state.

   In addition, the state machine uses two timers:

   ExpiryTimer (ET)
      This timer is restarted when a valid Join(*,*,RP) is received.
      Expiry of the ExpiryTimer causes the interface state to revert
      to NoInfo for this RP.

   Prune-Pending Timer (PPT)
      This timer is set when a valid Prune(*,*,RP) is received.
      Expiry of the Prune-Pending Timer causes the interface state
      to revert to NoInfo for this RP.

        Figure 2: Downstream per-interface (*,*,RP) state machine
                           in tabular form

		Event			
Prev State		Receive Join(*,*,RP)	Receive Prune (*,*,RP)	Prune-Pending Timer Expires	Expiry Timer Expires
NoInfo (NI)		-> J state start Expiry Timer	-> NI state	-	-
Join (J)		-> J state restart Expiry Timer	-> PP state start Prune-Pending Timer	-	-> NI state
Prune-Pending (PP)		-> J state restart Expiry Timer	-> PP state	-> NI state Send Prune-Echo(*,*,RP)	-> NI state

The transition events "Receive Join(*,*,RP)" and "Receive
Prune(*,*,RP)" imply receiving a Join or Prune targeted to this
router's primary IP address on the received interface. If the
upstream neighbor address field is not correct, these state
transitions in this state machine must not occur, although seeing
such a packet may cause state transitions in other state machines.

On unnumbered interfaces on point-to-point links, the router's
address should be the same as the source address it chose for the
Hello message it sent over that interface. However, on point-to-
point links we also recommend that for backwards compatibility PIM
Join/Prune messages with an upstream neighbor address field of all
zeros are also accepted.

Transitions from NoInfo State

When in NoInfo state, the following event may trigger a transition:

  Receive Join(*,*,RP)
        A Join(*,*,RP) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (*,*,RP) downstream state machine on interface I
        transitions to the Join state. The Expiry Timer (ET) is
        started and set to the HoldTime from the triggering Join/Prune
        message.

        Note that it is possible to receive a Join(*,*,RP) message for
        an RP for which we do not have information telling us that it
        is an RP. In the case of (*,*,RP) state, so long as we have a
        route to the RP, this will not cause a problem, and the
        transition should still take place.

Transitions from Join State

When in Join state, the following events may trigger a transition:

  Receive Join(*,*,RP)
        A Join(*,*,RP) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (*,*,RP) downstream state machine on interface I remains
        in Join state, and the Expiry Timer (ET) is restarted, set to
        maximum of its current value and the HoldTime from the
        triggering Join/Prune message.

Receive Prune(*,*,RP)
        A Prune(*,*,RP) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (*,*,RP) downstream state machine on interface I
        transitions to the Prune-Pending state. The Prune-Pending
        Timer is started. It is set to the J/P_Override_Interval(I)
        if the router has more than one neighbor on that interface;
        otherwise, it is set to zero, causing it to expire
        immediately.

   Expiry Timer Expires
        The Expiry Timer for the (*,*,RP) downstream state machine on
        interface I expires.

        The (*,*,RP) downstream state machine on interface I
        transitions to the NoInfo state.

   Transitions from Prune-Pending State

   When in Prune-Pending state, the following events may trigger a
   transition:

   Receive Join(*,*,RP)
        A Join(*,*,RP) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (*,*,RP) downstream state machine on interface I
        transitions to the Join state. The Prune-Pending Timer is
        canceled (without triggering an expiry event). The Expiry
        Timer is restarted, set to maximum of its current value and
        the HoldTime from the triggering Join/Prune message.

   Expiry Timer Expires
        The Expiry Timer for the (*,*,RP) downstream state machine on
        interface I expires.

        The (*,*,RP) downstream state machine on interface I
        transitions to the NoInfo state.

   Prune-Pending Timer Expires
        The Prune-Pending Timer for the (*,*,RP) downstream state
        machine on interface I expires.

        The (*,*,RP) downstream state machine on interface I
        transitions to the NoInfo state. A PruneEcho(*,*,RP) is sent
        onto the subnet connected to interface I.

The action "Send PruneEcho(*,*,RP)" is triggered when the
router stops forwarding on an interface as a result of a
prune. A PruneEcho(*,*,RP) is simply a Prune(*,*,RP) message
sent by the upstream router on a LAN with its own address in
the Upstream Neighbor Address field. Its purpose is to add
additional reliability so that if a Prune that should have
been overridden by another router is lost locally on the LAN,
then the PruneEcho may be received and cause the override to
happen. A PruneEcho(*,*,RP) need not be sent on an interface
that contains only a single PIM neighbor during the time this
state machine was in Prune-Pending state.

4.5.2. Receiving (*,G) Join/Prune Messages

   When a router receives a Join(*,G), it must first check to see
   whether the RP in the message matches RP(G) (the router's idea of who
   the RP is). If the RP in the message does not match RP(G), the
   Join(*,G) should be silently dropped. (Note that other source list
   entries, such as (S,G,rpt) or (S,G), in the same Group-Specific Set
   should still be processed.) If a router has no RP information (e.g.,
   has not recently received a BSR message), then it may choose to
   accept Join(*,G) and treat the RP in the message as RP(G). Received
   Prune(*,G) messages are processed even if the RP in the message does
   not match RP(G).

   The per-interface state machine for receiving (*,G) Join/Prune
   Messages is given below. There are three states:

     NoInfo (NI)
         The interface has no (*,G) Join state and no timers running.

     Join (J)
         The interface has (*,G) Join state, which will cause the
         router to forward packets destined for G from this interface
         except if there is also (S,G,rpt) prune information (see
         Section 4.5.4) or the router lost an assert on this interface.

     Prune-Pending (PP)
         The router has received a Prune(*,G) on this interface from a
         downstream neighbor and is waiting to see whether the prune
         will be overridden by another downstream router. For
         forwarding purposes, the Prune-Pending state functions exactly
         like the Join state.

In addition, the state machine uses two timers:

   Expiry Timer (ET)
        This timer is restarted when a valid Join(*,G) is received.
        Expiry of the Expiry Timer causes the interface state to
        revert to NoInfo for this group.

   Prune-Pending Timer (PPT)
        This timer is set when a valid Prune(*,G) is received. Expiry
        of the Prune-Pending Timer causes the interface state to
        revert to NoInfo for this group.

 Figure 3: Downstream per-interface (*,G) state machine in tabular form

Prev State		Receive Join(*,G)	Receive Prune(*,G)	Prune-Pending Timer Expires	Expiry Timer Expires
NoInfo (NI)		-> J state start Expiry Timer	-> NI state	-	-
Join (J)		-> J state restart Expiry Timer	-> PP state start Prune-Pending Timer	-	-> NI state
Prune-Pending (PP)		-> J state restart Expiry Timer	-> PP state	-> NI state Send Prune-Echo(*,G)	-> NI state

   The transition events "Receive Join(*,G)" and "Receive Prune(*,G)"
   imply receiving a Join or Prune targeted to this router's primary IP
   address on the received interface. If the upstream neighbor address
   field is not correct, these state transitions in this state machine
   must not occur, although seeing such a packet may cause state
   transitions in other state machines.

   On unnumbered interfaces on point-to-point links, the router's
   address should be the same as the source address it chose for the
   Hello message it sent over that interface. However, on point-to-

point links we also recommend that for backwards compatibility PIM
Join/Prune messages with an upstream neighbor address field of all
zeros are also accepted.

Transitions from NoInfo State

When in NoInfo state, the following event may trigger a transition:

   Receive Join(*,G)
        A Join(*,G) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (*,G) downstream state machine on interface I transitions
        to the Join state. The Expiry Timer (ET) is started and set
        to the HoldTime from the triggering Join/Prune message.

Transitions from Join State

When in Join state, the following events may trigger a transition:

   Receive Join(*,G)
        A Join(*,G) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (*,G) downstream state machine on interface I remains in
        Join state, and the Expiry Timer (ET) is restarted, set to
        maximum of its current value and the HoldTime from the
        triggering Join/Prune message.

   Receive Prune(*,G)
        A Prune(*,G) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (*,G) downstream state machine on interface I transitions
        to the Prune-Pending state. The Prune-Pending Timer is
        started. It is set to the J/P_Override_Interval(I) if the
        router has more than one neighbor on that interface;
        otherwise, it is set to zero, causing it to expire
        immediately.

   Expiry Timer Expires
        The Expiry Timer for the (*,G) downstream state machine on
        interface I expires.

        The (*,G) downstream state machine on interface I transitions
        to the NoInfo state.

    Transitions from Prune-Pending State

    When in Prune-Pending state, the following events may trigger a
    transition:

      Receive Join(*,G)
           A Join(*,G) is received on interface I with its Upstream
           Neighbor Address set to the router's primary IP address on I.

           The (*,G) downstream state machine on interface I transitions
           to the Join state. The Prune-Pending Timer is canceled
           (without triggering an expiry event). The Expiry Timer is
           restarted, set to maximum of its current value and the
           HoldTime from the triggering Join/Prune message.

      Expiry Timer Expires
           The Expiry Timer for the (*,G) downstream state machine on
           interface I expires.

           The (*,G) downstream state machine on interface I transitions
           to the NoInfo state.

      Prune-Pending Timer Expires
           The Prune-Pending Timer for the (*,G) downstream state machine
           on interface I expires.

           The (*,G) downstream state machine on interface I transitions
           to the NoInfo state. A PruneEcho(*,G) is sent onto the subnet
           connected to interface I.

           The action "Send PruneEcho(*,G)" is triggered when the router
           stops forwarding on an interface as a result of a prune. A
           PruneEcho(*,G) is simply a Prune(*,G) message sent by the
           upstream router on a LAN with its own address in the Upstream
           Neighbor Address field. Its purpose is to add additional
           reliability so that if a Prune that should have been
           overridden by another router is lost locally on the LAN, then
           the PruneEcho may be received and cause the override to
           happen. A PruneEcho(*,G) need not be sent on an interface
           that contains only a single PIM neighbor during the time this
           state machine was in Prune-Pending state.

4.5.3. Receiving (S,G) Join/Prune Messages

   The per-interface state machine for receiving (S,G) Join/Prune
   messages is given below and is almost identical to that for (*,G)
   messages. There are three states:

     NoInfo (NI)
          The interface has no (S,G) Join state and no (S,G) timers
          running.

     Join (J)
          The interface has (S,G) Join state, which will cause the
          router to forward packets from S destined for G from this
          interface if the (S,G) state is active (the SPTbit is set)
          except if the router lost an assert on this interface.

     Prune-Pending (PP)
          The router has received a Prune(S,G) on this interface from a
          downstream neighbor and is waiting to see whether the prune
          will be overridden by another downstream router. For
          forwarding purposes, the Prune-Pending state functions exactly
          like the Join state.

   In addition, there are two timers:

     Expiry Timer (ET)
          This timer is set when a valid Join(S,G) is received. Expiry
          of the Expiry Timer causes this state machine to revert to
          NoInfo state.

     Prune-Pending Timer (PPT)
          This timer is set when a valid Prune(S,G) is received. Expiry
          of the Prune-Pending Timer causes this state machine to revert
          to NoInfo state.

Figure 4: Downstream per-interface (S,G) state machine in tabular form

		Event			
Prev State		Receive Join(S,G)	Receive Prune(S,G)	Prune-Pending Timer Expires	Expiry Timer Expires
NoInfo (NI)		-> J state start Expiry Timer	-> NI state	-	-
Join (J)		-> J state restart Expiry Timer	-> PP state start Prune-Pending Timer	-	-> NI state
Prune-Pending (PP)		-> J state restart Expiry Timer	-> PP state	-> NI state Send Prune-Echo(S,G)	-> NI state

The transition events "Receive Join(S,G)" and "Receive Prune(S,G)" imply receiving a Join or Prune targeted to this router's primary IP address on the received interface. If the upstream neighbor address field is not correct, these state transitions in this state machine must not occur, although seeing such a packet may cause state transitions in other state machines.

On unnumbered interfaces on point-to-point links, the router's address should be the same as the source address it chose for the Hello message it sent over that interface. However, on point-to-point links we also recommend that for backwards compatibility PIM Join/Prune messages with an upstream neighbor address field of all zeros are also accepted.

Transitions from NoInfo State

When in NoInfo state, the following event may trigger a transition:

   Receive Join(S,G)
        A Join(S,G) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

           The (S,G) downstream state machine on interface I transitions
           to the Join state. The Expiry Timer (ET) is started and set
           to the HoldTime from the triggering Join/Prune message.

     Transitions from Join State

     When in Join state, the following events may trigger a transition:

       Receive Join(S,G)
           A Join(S,G) is received on interface I with its Upstream
           Neighbor Address set to the router's primary IP address on I.

           The (S,G) downstream state machine on interface I remains in
           Join state, and the Expiry Timer (ET) is restarted, set to
           maximum of its current value and the HoldTime from the
           triggering Join/Prune message.

       Receive Prune(S,G)
           A Prune(S,G) is received on interface I with its Upstream
           Neighbor Address set to the router's primary IP address on I.

           The (S,G) downstream state machine on interface I transitions
           to the Prune-Pending state. The Prune-Pending Timer is
           started. It is set to the J/P_Override_Interval(I) if the
           router has more than one neighbor on that interface;
           otherwise, it is set to zero, causing it to expire
           immediately.

       Expiry Timer Expires
           The Expiry Timer for the (S,G) downstream state machine on
           interface I expires.

           The (S,G) downstream state machine on interface I transitions
           to the NoInfo state.

     Transitions from Prune-Pending State

     When in Prune-Pending state, the following events may trigger a
     transition:

       Receive Join(S,G)
           A Join(S,G) is received on interface I with its Upstream
           Neighbor Address set to the router's primary IP address on I.

The (S,G) downstream state machine on interface I transitions
to the Join state. The Prune-Pending Timer is canceled
(without triggering an expiry event). The Expiry Timer is
restarted, set to maximum of its current value and the
HoldTime from the triggering Join/Prune message.

Expiry Timer Expires
     The Expiry Timer for the (S,G) downstream state machine on
     interface I expires.

     The (S,G) downstream state machine on interface I transitions
     to the NoInfo state.

Prune-Pending Timer Expires
     The Prune-Pending Timer for the (S,G) downstream state machine
     on interface I expires.

     The (S,G) downstream state machine on interface I transitions
     to the NoInfo state. A PruneEcho(S,G) is sent onto the subnet
     connected to interface I.

     The action "Send PruneEcho(S,G)" is triggered when the router
     stops forwarding on an interface as a result of a prune. A
     PruneEcho(S,G) is simply a Prune(S,G) message sent by the
     upstream router on a LAN with its own address in the Upstream
     Neighbor Address field. Its purpose is to add additional
     reliability so that if a Prune that should have been
     overridden by another router is lost locally on the LAN, then
     the PruneEcho may be received and cause the override to
     happen. A PruneEcho(S,G) need not be sent on an interface
     that contains only a single PIM neighbor during the time this
     state machine was in Prune-Pending state.

4.5.4. Receiving (S,G,rpt) Join/Prune Messages

The per-interface state machine for receiving (S,G,rpt) Join/Prune
messages is given below. There are five states:

  NoInfo (NI)
       The interface has no (S,G,rpt) Prune state and no (S,G,rpt)
       timers running.

  Prune (P)
       The interface has (S,G,rpt) Prune state, which will cause the
       router not to forward packets from S destined for G from this
       interface even though the interface has active (*,G) Join
       state.

   Prune-Pending (PP)
        The router has received a Prune(S,G,rpt) on this interface
        from a downstream neighbor and is waiting to see whether the
        prune will be overridden by another downstream router. For
        forwarding purposes, the Prune-Pending state functions exactly
        like the NoInfo state.

   PruneTmp (P')
        This state is a transient state that for forwarding purposes
        behaves exactly like the Prune state. A (*,G) Join has been
        received (which may cancel the (S,G,rpt) Prune). As we parse
        the Join/Prune message from top to bottom, we first enter this
        state if the message contains a (*,G) Join. Later in the
        message, we will normally encounter an (S,G,rpt) prune to
        reinstate the Prune state. However, if we reach the end of
        the message without encountering such a (S,G,rpt) prune, then
        we will revert to NoInfo state in this state machine.

        As no time is spent in this state, no timers can expire.

   Prune-Pending-Tmp (PP')
        This state is a transient state that is identical to P' except
        that it is associated with the PP state rather than the P
        state. For forwarding purposes, PP' behaves exactly like PP
        state.

   In addition, there are two timers:

   Expiry Timer (ET)
        This timer is set when a valid Prune(S,G,rpt) is received.
        Expiry of the Expiry Timer causes this state machine to revert
        to NoInfo state.

   Prune-Pending Timer (PPT)
        This timer is set when a valid Prune(S,G,rpt) is received.
        Expiry of the Prune-Pending Timer causes this state machine to
        move on to Prune state.

Figure 5: Downstream per-interface (S,G,rpt) state machine
in tabular form

		Event
Prev State		Receive Join(*,G)	Receive Join (S,G,rpt)	Receive Prune (S,G,rpt)	End of Message	Prune-Pending Timer Expires	Expiry Timer Expires
NoInfo (NI)		-	-	-> PP state start Prune-Pending Timer; start Expiry Timer	-	-	-
Prune (P)		-> P' state	-> NI state	-> P state restart Expiry Timer	-	-	-> NI state
Prune-Pending (PP)		-> PP' state	-> NI state	-	-	-> P state	-
PruneTmp (P')		-	-	-> P state restart Expiry Timer	-> NI state	-	-
Prune-Pending-Tmp (PP')		-	-	-> PP state restart Expiry Timer	-> NI state	-	-

The transition events "Receive Join(S,G,rpt)", "Receive
Prune(S,G,rpt)", and "Receive Join(*,G)" imply receiving a Join or
Prune targeted to this router's primary IP address on the received
interface. If the upstream neighbor address field is not correct,

these state transitions in this state machine must not occur,
although seeing such a packet may cause state transitions in other
state machines.

On unnumbered interfaces on point-to-point links, the router's
address should be the same as the source address it chose for the
Hello message it sent over that interface. However, on point-to-
point links we also recommend that PIM Join/Prune messages with an
upstream neighbor address field of all zeros are also accepted.

Transitions from NoInfo State

When in NoInfo (NI) state, the following event may trigger a
transition:

  Receive Prune(S,G,rpt)
        A Prune(S,G,rpt) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (S,G,rpt) downstream state machine on interface I
        transitions to the Prune-Pending state. The Expiry Timer (ET)
        is started and set to the HoldTime from the triggering
        Join/Prune message. The Prune-Pending Timer is started. It
        is set to the J/P_Override_Interval(I) if the router has more
        than one neighbor on that interface; otherwise, it is set to
        zero, causing it to expire immediately.

Transitions from Prune-Pending State

When in Prune-Pending (PP) state, the following events may trigger a
transition:

  Receive Join(*,G)
        A Join(*,G) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (S,G,rpt) downstream state machine on interface I
        transitions to Prune-Pending-Tmp state whilst the remainder of
        the compound Join/Prune message containing the Join(*,G) is
        processed.

  Receive Join(S,G,rpt)
        A Join(S,G,rpt) is received on interface I with its Upstream
        Neighbor Address set to the router's primary IP address on I.

        The (S,G,rpt) downstream state machine on interface I
        transitions to NoInfo state. ET and PPT are canceled.

   Prune-Pending Timer Expires
         The Prune-Pending Timer for the (S,G,rpt) downstream state
         machine on interface I expires.

         The (S,G,rpt) downstream state machine on interface I
         transitions to the Prune state.

   Transitions from Prune State

   When in Prune (P) state, the following events may trigger a
   transition:

     Receive Join(*,G)
           A Join(*,G) is received on interface I with its Upstream
           Neighbor Address set to the router's primary IP address on I.

           The (S,G,rpt) downstream state machine on interface I
           transitions to PruneTmp state whilst the remainder of the
           compound Join/Prune message containing the Join(*,G) is
           processed.

     Receive Join(S,G,rpt)
           A Join(S,G,rpt) is received on interface I with its Upstream
           Neighbor Address set to the router's primary IP address on I.

           The (S,G,rpt) downstream state machine on interface I
           transitions to NoInfo state. ET and PPT are canceled.

     Receive Prune(S,G,rpt)
           A Prune(S,G,rpt) is received on interface I with its Upstream
           Neighbor Address set to the router's primary IP address on I.

           The (S,G,rpt) downstream state machine on interface I remains
           in Prune state. The Expiry Timer (ET) is restarted, set to
           maximum of its current value and the HoldTime from the
           triggering Join/Prune message.

     Expiry Timer Expires
           The Expiry Timer for the (S,G,rpt) downstream state machine on
           interface I expires.

           The (S,G,rpt) downstream state machine on interface I
           transitions to the NoInfo state.

   Transitions from Prune-Pending-Tmp State

   When in Prune-Pending-Tmp (PP') state and processing a compound
   Join/Prune message, the following events may trigger a transition:

   Receive Prune(S,G,rpt)
        The compound Join/Prune message contains a Prune(S,G,rpt).

        The (S,G,rpt) downstream state machine on interface I
        transitions back to the Prune-Pending state. The Expiry Timer
        (ET) is restarted, set to maximum of its current value and the
        HoldTime from the triggering Join/Prune message.

   End of Message
        The end of the compound Join/Prune message is reached.

        The (S,G,rpt) downstream state machine on interface I
        transitions to the NoInfo state. ET and PPT are canceled.

   Transitions from PruneTmp State

   When in PruneTmp (P') state and processing a compound Join/Prune
   message, the following events may trigger a transition:

   Receive Prune(S,G,rpt)
        The compound Join/Prune message contains a Prune(S,G,rpt).

        The (S,G,rpt) downstream state machine on interface I
        transitions back to the Prune state. The Expiry Timer (ET) is
        restarted, set to maximum of its current value and the
        HoldTime from the triggering Join/Prune message.

   End of Message
        The end of the compound Join/Prune message is reached.

        The (S,G,rpt) downstream state machine on interface I
        transitions to the NoInfo state. ET is canceled.

   Notes:

   Receiving a Prune(*,G) does not affect the (S,G,rpt) downstream state
   machine.

   Receiving a Join(*,*,RP) does not affect the (S,G,rpt) downstream
   state machine. If a router has originated Join(*,*,RP) and pruned a
   source off it using Prune(S,G,rpt), then to receive that source again
   it should explicitly re-join using Join(S,G,rpt) or Join(*,G). In
   some LAN topologies it is possible for a router sending a new
   Join(*,*,RP) to have to wait as much as a Join/Prune Interval before
   noticing that it needs to override a neighbor's preexisting
   Prune(S,G,rpt). This is considered acceptable, as (*,*,RP) state is
   intended to be used only in long-lived and persistent scenarios.

4.5.5. Sending (*,*,RP) Join/Prune Messages

   The per-interface state machines for (*,*,RP) hold join state from
   downstream PIM routers. This state then determines whether a router
   needs to propagate a Join(*,*,RP) upstream towards the RP.

   If a router wishes to propagate a Join(*,*,RP) upstream, it must also
   watch for messages on its upstream interface from other routers on
   that subnet, and these may modify its behavior. If it sees a
   Join(*,*,RP) to the correct upstream neighbor, it should suppress its
   own Join(*,*,RP). If it sees a Prune(*,*,RP) to the correct upstream
   neighbor, it should be prepared to override that prune by sending a
   Join(*,*,RP) almost immediately. Finally, if it sees the Generation
   ID (see Section 4.3) of the correct upstream neighbor change, it
   knows that the upstream neighbor has lost state, and it should be
   prepared to refresh the state by sending a Join(*,*,RP) almost
   immediately.

   In addition, if the MRIB changes to indicate that the next hop
   towards the RP has changed, the router should prune off from the old
   next hop and join towards the new next hop.

   The upstream (*,*,RP) state machine contains only two states:

   Not Joined
      The downstream state machines and local membership information do
      not indicate that the router needs to join the (*,*,RP) tree for
      this RP.

   Joined
      The downstream state machines and local membership information
      indicate that the router should join the (*,*,RP) tree for this
      RP.

   In addition, one timer JT(*,*,RP) is kept that is used to trigger the
   sending of a Join(*,*,RP) to the upstream next hop towards the RP,
   NBR(RPF_interface(RP), MRIB.next_hop(RP)).

Figure 6: Upstream (*,*,RP) state machine in tabular form

```
+------------------++-------------------------------------------------+
|                  ||                       Event                      |
|   Prev State     ++------------------------+------------------------+
|                  ||  JoinDesired           |  JoinDesired           |
|                  ||  (*,*,RP) ->True       |  (*,*,RP) ->False      |
+------------------++------------------------+------------------------+
|                  ||  -> J state            |  -                     |
|  NotJoined (NJ)  ||  Send Join(*,*,RP);    |                        |
|                  ||  Set Join Timer to     |                        |
|                  ||  t_periodic            |                        |
+------------------++------------------------+------------------------+
|  Joined (J)      ||  -                     |  -> NJ state           |
|                  ||                        |  Send Prune            |
|                  ||                        |  (*,*,RP); Cancel      |
|                  ||                        |  Join Timer            |
+------------------++------------------------+------------------------+
```

In addition, we have the following transitions, which occur within the Joined state:

```
+--------------------------------------------------------------------+
|                          In Joined (J) State                        |
+------------------+------------------+----------------------------+
|  Timer Expires   |  See             |  See                       |
|                  |  Join(*,*,RP)    |  Prune(*,*,RP)             |
|                  |  to MRIB.        |  to MRIB.                  |
|                  |  next_hop(RP)    |  next_hop(RP)             |
+------------------+------------------+----------------------------+
|  Send            |  Increase Join   |  Decrease Join            |
|  Join(*,*,RP);   |  Timer to        |  Timer to                 |
|  Set Join Timer  |  t_joinsuppress  |  t_override               |
|  to t_periodic   |                  |                           |
+------------------+------------------+----------------------------+
```

```
+------------------------------------------------------------------------+
|                          In Joined (J) State                           |
+-------------------------------------+----------------------------------+
|      NBR(RPF_interface(RP),         |     MRIB.next_hop(RP) GenID       |
|      MRIB.next_hop(RP))             |     changes                       |
|      changes                        |                                   |
+-------------------------------------+----------------------------------+
|      Send Join(*,*,RP) to new       |     Decrease Join Timer to        |
|      next hop; Send                 |     t_override                    |
|      Prune(*,*,RP) to old           |                                   |
|      next hop; set Join Timer       |                                   |
|      to t_periodic                  |                                   |
+-------------------------------------+----------------------------------+
```

   This state machine uses the following macro:

```
      bool JoinDesired(*,*,RP) {
          if immediate_olist(*,*,RP) != NULL
              return TRUE
          else
              return FALSE
      }
```

   JoinDesired(*,*,RP) is true when the router has received (*,*,RP)
   Joins from any downstream interface. Note that although JoinDesired
   is true, the router's sending of a Join(*,*,RP) message may be
   suppressed by another router sending a Join(*,*,RP) onto the upstream
   interface.

   Transitions from NotJoined State

   When the upstream (*,*,RP) state machine is in NotJoined state, the
   following event may trigger a state transition:

     JoinDesired(*,*,RP) becomes True
          The downstream state for (*,*,RP) has changed so that at least
          one interface is in immediate_olist(*,*,RP), making
          JoinDesired(*,*,RP) become True.

          The upstream (*,*,RP) state machine transitions to Joined
          state. Send Join(*,*,RP) to the appropriate upstream
          neighbor, which is NBR(RPF_interface(RP), MRIB.next_hop(RP)).
          Set the Join Timer (JT) to expire after t_periodic seconds.

   Transitions from Joined State

   When the upstream (*,*,RP) state machine is in Joined state, the
   following events may trigger state transitions:

JoinDesired(*,*,RP) becomes False
     The downstream state for (*,*,RP) has changed so no interface
     is in immediate_olist(*,*,RP), making JoinDesired(*,*,RP)
     become False.

     The upstream (*,*,RP) state machine transitions to NotJoined
     state. Send Prune(*,*,RP) to the appropriate upstream
     neighbor, which is NBR(RPF_interface(RP), MRIB.next_hop(RP)).
     Cancel the Join Timer (JT).

Join Timer Expires
     The Join Timer (JT) expires, indicating time to send a
     Join(*,*,RP)

     Send Join(*,*,RP) to the appropriate upstream neighbor, which
     is NBR(RPF_interface(RP), MRIB.next_hop(RP)). Restart the
     Join Timer (JT) to expire after t_periodic seconds.

See Join(*,*,RP) to MRIB.next_hop(RP)
     This event is only relevant if RPF_interface(RP) is a shared
     medium. This router sees another router on RPF_interface(RP)
     send a Join(*,*,RP) to NBR(RPF_interface(RP),
     MRIB.next_hop(RP)). This causes this router to suppress its
     own Join.

     The upstream (*,*,RP) state machine remains in Joined state.

     Let t_joinsuppress be the minimum of t_suppressed and the
     HoldTime from the Join/Prune message triggering this event.
     If the Join Timer is set to expire in less than t_joinsuppress
     seconds, reset it so that it expires after t_joinsuppress
     seconds. If the Join Timer is set to expire in more than
     t_joinsuppress seconds, leave it unchanged.

See Prune(*,*,RP) to MRIB.next_hop(RP)
     This event is only relevant if RPF_interface(RP) is a shared
     medium. This router sees another router on RPF_interface(RP)
     send a Prune(*,*,RP) to NBR(RPF_interface(RP),
     MRIB.next_hop(RP)). As this router is in Joined state, it
     must override the Prune after a short random interval.

     The upstream (*,*,RP) state machine remains in Joined state.
     If the Join Timer is set to expire in more than t_override
     seconds, reset it so that it expires after t_override seconds.
     If the Join Timer is set to expire in less than t_override
     seconds, leave it unchanged.

        NBR(RPF_interface(RP), MRIB.next_hop(RP)) changes
             A change in the MRIB routing base causes the next hop towards
             the RP to change.

             The upstream (*,*,RP) state machine remains in Joined state.
             Send Join(*,*,RP) to the new upstream neighbor, which is the
             new value of NBR(RPF_interface(RP), MRIB.next_hop(RP)). Send
             Prune(*,*,RP) to the old upstream neighbor, which is the old
             value of NBR(RPF_interface(RP), MRIB.next_hop(RP)). Set the
             Join Timer (JT) to expire after t_periodic seconds.

        MRIB.next_hop(RP) GenID changes
             The Generation ID of the router that is MRIB.next_hop(RP)
             changes. This normally means that this neighbor has lost
             state, and so the state must be refreshed.

             The upstream (*,*,RP) state machine remains in Joined state.
             If the Join Timer is set to expire in more than t_override
             seconds, reset it so that it expires after t_override seconds.

4.5.6. Sending (*,G) Join/Prune Messages

   The per-interface state machines for (*,G) hold join state from
   downstream PIM routers. This state then determines whether a router
   needs to propagate a Join(*,G) upstream towards the RP.

   If a router wishes to propagate a Join(*,G) upstream, it must also
   watch for messages on its upstream interface from other routers on
   that subnet, and these may modify its behavior. If it sees a
   Join(*,G) to the correct upstream neighbor, it should suppress its
   own Join(*,G). If it sees a Prune(*,G) to the correct upstream
   neighbor, it should be prepared to override that prune by sending a
   Join(*,G) almost immediately. Finally, if it sees the Generation ID
   (see Section 4.3) of the correct upstream neighbor change, it knows
   that the upstream neighbor has lost state, and it should be prepared
   to refresh the state by sending a Join(*,G) almost immediately.

   If a (*,G) Assert occurs on the upstream interface, and this changes
   this router's idea of the upstream neighbor, it should be prepared to
   ensure that the Assert winner is aware of downstream routers by
   sending a Join(*,G) almost immediately.

   In addition, if the MRIB changes to indicate that the next hop
   towards the RP has changed, and either the upstream interface changes
   or there is no Assert winner on the upstream interface, the router
   should prune off from the old next hop and join towards the new next
   hop.

The upstream (*,G) state machine only contains two states:

Not Joined
    The downstream state machines indicate that the router does not
    need to join the RP tree for this group.

Joined
    The downstream state machines indicate that the router should join
    the RP tree for this group.

In addition, one timer JT(*,G) is kept that is used to trigger the
sending of a Join(*,G) to the upstream next hop towards the RP,
RPF'(*,G).

Figure 7: Upstream (*,G) state machine in tabular form

```
+------------------++--------------------------------------------------+
|                  ||                     Event                        |
|   Prev State     ++-----------------------+--------------------------+
|                  ||   JoinDesired(*,G)     |   JoinDesired(*,G)       |
|                  ||   ->True               |   ->False                |
+------------------++-----------------------+--------------------------+
|                  ||   -> J state           |   -                      |
|   NotJoined (NJ) ||   Send Join(*,G);      |                          |
|                  ||   Set Join Timer to    |                          |
|                  ||   t_periodic           |                          |
+------------------++-----------------------+--------------------------+
|   Joined (J)     ||   -                    |   -> NJ state            |
|                  ||                        |   Send Prune(*,G);       |
|                  ||                        |   Cancel Join Timer      |
+------------------++-----------------------+--------------------------+
```

In addition, we have the following transitions, which occur within
the Joined state:

```
+----------------------------------------------------------------------+
|                        In Joined (J) State                           |
+---------------+----------------+----------------+--------------------+
|Timer Expires  | See Join(*,G)  | See Prune(*,G) | RPF'(*,G)          |
|               | to RPF'(*,G)   | to RPF'(*,G)   | changes due to     |
|               |                |                | an Assert          |
+---------------+----------------+----------------+--------------------+
|Send           | Increase Join  | Decrease Join  | Decrease Join      |
|Join(*,G); Set | Timer to       | Timer to       | Timer to           |
|Join Timer to  | t_joinsuppress | t_override     | t_override         |
|t_periodic     |                |                |                    |
+---------------+----------------+----------------+--------------------+
```

```
+------------------------------------------------------------------+
|                     In Joined (J) State                          |
+----------------------------------+-------------------------------+
|     RPF'(*,G) changes not        |     RPF'(*,G) GenID changes   |
|     due to an Assert             |                               |
+----------------------------------+-------------------------------+
|     Send Join(*,G) to new        |     Decrease Join Timer to    |
|     next hop; Send               |     t_override                |
|     Prune(*,G) to old next       |                               |
|     hop; Set Join Timer to       |                               |
|     t_periodic                   |                               |
+----------------------------------+-------------------------------+
```

This state machine uses the following macro:

```
bool JoinDesired(*,G) {
    if (immediate_olist(*,G) != NULL OR
        (JoinDesired(*,*,RP(G)) AND
         AssertWinner(*, G, RPF_interface(RP(G))) != NULL))
        return TRUE
    else
        return FALSE
}
```

JoinDesired(*,G) is true when the router has forwarding state that
would cause it to forward traffic for G using shared tree state.
Note that although JoinDesired is true, the router's sending of a
Join(*,G) message may be suppressed by another router sending a
Join(*,G) onto the upstream interface.

Transitions from NotJoined State

When the upstream (*,G) state machine is in NotJoined state, the
following event may trigger a state transition:

  JoinDesired(*,G) becomes True
        The macro JoinDesired(*,G) becomes True, e.g., because the
        downstream state for (*,G) has changed so that at least one
        interface is in immediate_olist(*,G).

        The upstream (*,G) state machine transitions to Joined state.
        Send Join(*,G) to the appropriate upstream neighbor, which is
        RPF'(*,G). Set the Join Timer (JT) to expire after t_periodic
        seconds.

Transitions from Joined State

When the upstream (*,G) state machine is in Joined state, the
following events may trigger state transitions:

  JoinDesired(*,G) becomes False
        The macro JoinDesired(*,G) becomes False, e.g., because the
        downstream state for (*,G) has changed so no interface is in
        immediate_olist(*,G).

        The upstream (*,G) state machine transitions to NotJoined
        state. Send Prune(*,G) to the appropriate upstream neighbor,
        which is RPF'(*,G). Cancel the Join Timer (JT).

  Join Timer Expires
        The Join Timer (JT) expires, indicating time to send a
        Join(*,G)

        Send Join(*,G) to the appropriate upstream neighbor, which is
        RPF'(*,G). Restart the Join Timer (JT) to expire after
        t_periodic seconds.

  See Join(*,G) to RPF'(*,G)
        This event is only relevant if RPF_interface(RP(G)) is a
        shared medium. This router sees another router on
        RPF_interface(RP(G)) send a Join(*,G) to RPF'(*,G). This
        causes this router to suppress its own Join.

        The upstream (*,G) state machine remains in Joined state.

        Let t_joinsuppress be the minimum of t_suppressed and the
        HoldTime from the Join/Prune message triggering this event.
        If the Join Timer is set to expire in less than t_joinsuppress
        seconds, reset it so that it expires after t_joinsuppress
        seconds. If the Join Timer is set to expire in more than
        t_joinsuppress seconds, leave it unchanged.

  See Prune(*,G) to RPF'(*,G)
        This event is only relevant if RPF_interface(RP(G)) is a
        shared medium. This router sees another router on
        RPF_interface(RP(G)) send a Prune(*,G) to RPF'(*,G). As this
        router is in Joined state, it must override the Prune after a
        short random interval.

The upstream (*,G) state machine remains in Joined state. If
the Join Timer is set to expire in more than t_override
seconds, reset it so that it expires after t_override seconds.
If the Join Timer is set to expire in less than t_override
seconds, leave it unchanged.

RPF'(*,G) changes due to an Assert
     The current next hop towards the RP changes due to an
     Assert(*,G) on the RPF_interface(RP(G)).

     The upstream (*,G) state machine remains in Joined state. If
     the Join Timer is set to expire in more than t_override
     seconds, reset it so that it expires after t_override seconds.
     If the Join Timer is set to expire in less than t_override
     seconds, leave it unchanged.

RPF'(*,G) changes not due to an Assert
     An event occurred that caused the next hop towards the RP for
     G to change. This may be caused by a change in the MRIB
     routing database or the group-to-RP mapping. Note that this
     transition does not occur if an Assert is active and the
     upstream interface does not change.

     The upstream (*,G) state machine remains in Joined state.
     Send Join(*,G) to the new upstream neighbor, which is the new
     value of RPF'(*,G). Send Prune(*,G) to the old upstream
     neighbor, which is the old value of RPF'(*,G). Use the new
     value of RP(G) in the Prune(*,G) message or all zeros if RP(G)
     becomes unknown (old value of RP(G) may be used instead to
     improve behavior in routers implementing older versions of
     this spec). Set the Join Timer (JT) to expire after
     t_periodic seconds.

RPF'(*,G) GenID changes
     The Generation ID of the router that is RPF'(*,G) changes.
     This normally means that this neighbor has lost state, and so
     the state must be refreshed.

     The upstream (*,G) state machine remains in Joined state. If
     the Join Timer is set to expire in more than t_override
     seconds, reset it so that it expires after t_override seconds.

4.5.7. Sending (S,G) Join/Prune Messages

   The per-interface state machines for (S,G) hold join state from
   downstream PIM routers. This state then determines whether a router
   needs to propagate a Join(S,G) upstream towards the source.

   If a router wishes to propagate a Join(S,G) upstream, it must also
   watch for messages on its upstream interface from other routers on
   that subnet, and these may modify its behavior. If it sees a
   Join(S,G) to the correct upstream neighbor, it should suppress its
   own Join(S,G). If it sees a Prune(S,G), Prune(S,G,rpt), or
   Prune(*,G) to the correct upstream neighbor towards S, it should be
   prepared to override that prune by scheduling a Join(S,G) to be sent
   almost immediately. Finally, if it sees the Generation ID of its
   upstream neighbor change, it knows that the upstream neighbor has
   lost state, and it should refresh the state by scheduling a Join(S,G)
   to be sent almost immediately.

   If a (S,G) Assert occurs on the upstream interface, and this changes
   the this router's idea of the upstream neighbor, it should be
   prepared to ensure that the Assert winner is aware of downstream
   routers by scheduling a Join(S,G) to be sent almost immediately.

   In addition, if MRIB changes cause the next hop towards the source to
   change, and either the upstream interface changes or there is no
   Assert winner on the upstream interface, the router should send a
   prune to the old next hop and a join to the new next hop.

   The upstream (S,G) state machine only contains two states:

   Not Joined
      The downstream state machines and local membership information do
      not indicate that the router needs to join the shortest-path tree
      for this (S,G).

   Joined
      The downstream state machines and local membership information
      indicate that the router should join the shortest-path tree for
      this (S,G).

   In addition, one timer JT(S,G) is kept that is used to trigger the
   sending of a Join(S,G) to the upstream next hop towards S, RPF'(S,G).

Figure 8: Upstream (S,G) state machine in tabular form

```
+------------------+--------------------------------------------------+
|                  |                      Event                       |
|   Prev State     +-------------------------+------------------------+
|                  | JoinDesired(S,G)        | JoinDesired(S,G)       |
|                  | ->True                  | ->False                |
+------------------+-------------------------+------------------------+
| NotJoined (NJ)   | -> J state              | -                      |
|                  | Send Join(S,G);         |                        |
|                  | Set Join Timer to       |                        |
|                  | t_periodic              |                        |
+------------------+-------------------------+------------------------+
| Joined (J)       | -                       | -> NJ state            |
|                  |                         | Send Prune(S,G);       |
|                  |                         | Set SPTbit(S,G) to     |
|                  |                         | FALSE; Cancel Join     |
|                  |                         | Timer                  |
+------------------+-------------------------+------------------------+
```

In addition, we have the following transitions, which occur within the Joined state:

```
+---------------------------------------------------------------------+
|                        In Joined (J) State                          |
+----------------+----------------+----------------+------------------+
| Timer Expires  | See Join(S,G)  | See Prune(S,G) | See Prune        |
|                | to RPF'(S,G)   | to RPF'(S,G)   | (S,G,rpt) to     |
|                |                |                | RPF'(S,G)        |
+----------------+----------------+----------------+------------------+
| Send           | Increase Join  | Decrease Join  | Decrease Join    |
| Join(S,G); Set | Timer to       | Timer to       | Timer to         |
| Join Timer to  | t_joinsuppress | t_override     | t_override       |
| t_periodic     |                |                |                  |
+----------------+----------------+----------------+------------------+
```

```
+----------------------------------------------------------------------+
|                        In Joined (J) State                           |
+-----------------+-----------------+-----------------+-----------------+
| See Prune(*,G)  | RPF'(S,G)       | RPF'(S,G)       | RPF'(S,G)       |
| to RPF'(S,G)    | changes not     | GenID changes   | changes due to  |
|                 | due to an       |                 | an Assert       |
|                 | Assert          |                 |                 |
+-----------------+-----------------+-----------------+-----------------+
| Decrease Join   | Send Join(S,G)  | Decrease Join   | Decrease Join   |
| Timer to        | to new next     | Timer to        | Timer to        |
| t_override      | hop; Send       | t_override      | t_override      |
|                 | Prune(S,G) to   |                 |                 |
|                 | old next hop;   |                 |                 |
|                 | Set Join Timer  |                 |                 |
|                 | to t_periodic   |                 |                 |
+-----------------+-----------------+-----------------+-----------------+
```

This state machine uses the following macro:

```
bool JoinDesired(S,G) {
    return( immediate_olist(S,G) != NULL
            OR ( KeepaliveTimer(S,G) is running
                AND inherited_olist(S,G) != NULL ) )
}
```

JoinDesired(S,G) is true when the router has forwarding state that
would cause it to forward traffic for G using source tree state. The
source tree state can be as a result of either active source-specific
join state, or the (S,G) Keepalive Timer and active non-source-
specific state. Note that although JoinDesired is true, the router's
sending of a Join(S,G) message may be suppressed by another router
sending a Join(S,G) onto the upstream interface.

Transitions from NotJoined State

When the upstream (S,G) state machine is in NotJoined state, the
following event may trigger a state transition:

    JoinDesired(S,G) becomes True
          The macro JoinDesired(S,G) becomes True, e.g., because the
          downstream state for (S,G) has changed so that at least one
          interface is in inherited_olist(S,G).

          The upstream (S,G) state machine transitions to Joined state.
          Send Join(S,G) to the appropriate upstream neighbor, which is
          RPF'(S,G). Set the Join Timer (JT) to expire after t_periodic
          seconds.

   Transitions from Joined State

   When the upstream (S,G) state machine is in Joined state, the
   following events may trigger state transitions:

     JoinDesired(S,G) becomes False
          The macro JoinDesired(S,G) becomes False, e.g., because the
          downstream state for (S,G) has changed so no interface is in
          inherited_olist(S,G).

          The upstream (S,G) state machine transitions to NotJoined
          state. Send Prune(S,G) to the appropriate upstream neighbor,
          which is RPF'(S,G). Cancel the Join Timer (JT), and set
          SPTbit(S,G) to FALSE.

     Join Timer Expires
          The Join Timer (JT) expires, indicating time to send a
          Join(S,G)

          Send Join(S,G) to the appropriate upstream neighbor, which is
          RPF'(S,G). Restart the Join Timer (JT) to expire after
          t_periodic seconds.

     See Join(S,G) to RPF'(S,G)
          This event is only relevant if RPF_interface(S) is a shared
          medium. This router sees another router on RPF_interface(S)
          send a Join(S,G) to RPF'(S,G). This causes this router to
          suppress its own Join.

          The upstream (S,G) state machine remains in Joined state.

          Let t_joinsuppress be the minimum of t_suppressed and the
          HoldTime from the Join/Prune message triggering this event.

          If the Join Timer is set to expire in less than t_joinsuppress
          seconds, reset it so that it expires after t_joinsuppress
          seconds. If the Join Timer is set to expire in more than
          t_joinsuppress seconds, leave it unchanged.

     See Prune(S,G) to RPF'(S,G)
          This event is only relevant if RPF_interface(S) is a shared
          medium. This router sees another router on RPF_interface(S)
          send a Prune(S,G) to RPF'(S,G). As this router is in Joined
          state, it must override the Prune after a short random
          interval.

The upstream (S,G) state machine remains in Joined state. If
the Join Timer is set to expire in more than t_override
seconds, reset it so that it expires after t_override seconds.

See Prune(S,G,rpt) to RPF'(S,G)
     This event is only relevant if RPF_interface(S) is a shared
     medium. This router sees another router on RPF_interface(S)
     send a Prune(S,G,rpt) to RPF'(S,G). If the upstream router is
     an RFC-2362-compliant PIM router, then the Prune(S,G,rpt) will
     cause it to stop forwarding. For backwards compatibility,
     this router should override the prune so that forwarding
     continues.

     The upstream (S,G) state machine remains in Joined state. If
     the Join Timer is set to expire in more than t_override
     seconds, reset it so that it expires after t_override seconds.

See Prune(*,G) to RPF'(S,G)
     This event is only relevant if RPF_interface(S) is a shared
     medium. This router sees another router on RPF_interface(S)
     send a Prune(*,G) to RPF'(S,G). If the upstream router is an
     RFC-2362-compliant PIM router, then the Prune(*,G) will cause
     it to stop forwarding. For backwards compatibility, this
     router should override the prune so that forwarding continues.

     The upstream (S,G) state machine remains in Joined state. If
     the Join Timer is set to expire in more than t_override
     seconds, reset it so that it expires after t_override seconds.

RPF'(S,G) changes due to an Assert
     The current next hop towards S changes due to an Assert(S,G)
     on the RPF_interface(S).

     The upstream (S,G) state machine remains in Joined state. If
     the Join Timer is set to expire in more than t_override
     seconds, reset it so that it expires after t_override seconds.
     If the Join Timer is set to expire in less than t_override
     seconds, leave it unchanged.

RPF'(S,G) changes not due to an Assert
     An event occurred that caused the next hop towards S to
     change. Note that this transition does not occur if an Assert
     is active and the upstream interface does not change.

          The upstream (S,G) state machine remains in Joined state.
          Send Join(S,G) to the new upstream neighbor, which is the new
          value of RPF'(S,G). Send Prune(S,G) to the old upstream
          neighbor, which is the old value of RPF'(S,G). Set the Join
          Timer (JT) to expire after t_periodic seconds.

      RPF'(S,G) GenID changes
          The Generation ID of the router that is RPF'(S,G) changes.
          This normally means that this neighbor has lost state, and so
          the state must be refreshed.

          The upstream (S,G) state machine remains in Joined state. If
          the Join Timer is set to expire in more than t_override
          seconds, reset it so that it expires after t_override seconds.

4.5.8. (S,G,rpt) Periodic Messages

   (S,G,rpt) Joins and Prunes are (S,G) Joins or Prunes sent on the RP
   tree with the RPT bit set, either to modify the results of (*,G)
   Joins, or to override the behavior of other upstream LAN peers. The
   next section describes the rules for sending triggered messages.
   This section describes the rules for including a Prune(S,G,rpt)
   message with a Join(*,G).

   When a router is going to send a Join(*,G), it should use the
   following pseudocode, for each (S,G) for which it has state, to
   decide whether to include a Prune(S,G,rpt) in the compound Join/Prune
   message:

```
  if( SPTbit(S,G) == TRUE ) {
      # Note: If receiving (S,G) on the SPT, we only prune off the
      # shared tree if the RPF neighbors differ.
       if( RPF'(*,G) != RPF'(S,G) ) {
           add Prune(S,G,rpt) to compound message
       }
  } else if ( inherited_olist(S,G,rpt) == NULL ) {
    # Note: all (*,G) olist interfaces received RPT prunes for (S,G).
    add Prune(S,G,rpt) to compound message
  } else if ( RPF'(*,G) != RPF'(S,G,rpt) {
    # Note: we joined the shared tree, but there was an (S,G) assert
    # and the source tree RPF neighbor is different.
    add Prune(S,G,rpt) to compound message
  }
```

   Note that Join(S,G,rpt) is normally sent not as a periodic message,
   but only as a triggered message.

4.5.9. State Machine for (S,G,rpt) Triggered Messages

   The state machine for (S,G,rpt) triggered messages is required per-
   (S,G) when there is (*,G) or (*,*,RP) join state at a router, and the
   router or any of its upstream LAN peers wishes to prune S off the RP
   tree.

   There are three states in the state machine. One of the states is
   when there is neither (*,G) nor (*,*,RP(G)) join state at this
   router. If there is (*,G) or (*,*,RP(G)) join state at the router,
   then the state machine must be at one of the other two states. The
   three states are:

   Pruned(S,G,rpt)
      (*,G) or (*,*,RP(G)) Joined, but (S,G,rpt) pruned

   NotPruned(S,G,rpt)
      (*,G) or (*,*,RP(G)) Joined, and (S,G,rpt) not pruned

   RPTNotJoined(G)
      neither (*,G) nor (*,*,RP(G)) has been joined.

   In addition, there is an (S,G,rpt) Override Timer, OT(S,G,rpt), which
   is used to delay triggered Join(S,G,rpt) messages to prevent
   implosions of triggered messages.

    Figure 9: Upstream (S,G,rpt) state machine for triggered messages
                            in tabular form

```
+------------++----------------------------------------------------------+
|            ||                          Event                           |
|            ++--------------+-------------+------------+-----------------+
|Prev State  || PruneDesired | PruneDesired| RPTJoin    | inherited_      |
|            || (S,G,rpt)    | (S,G,rpt)   | Desired(G) | olist           |
|            || ->True       | ->False     | ->False    | (S,G,rpt)       |
|            ||              |             |            | ->non-NULL      |
+------------++--------------+-------------+------------+-----------------+
|RPTNotJoined|| -> P state   | -           | -          | -> NP state     |
|(G) (NJ)    ||              |             |            |                 |
+------------++--------------+-------------+------------+-----------------+
|Pruned      || -            | -> NP state | -> NJ state| -               |
|(S,G,rpt)   ||              | Send Join   |            |                 |
|(P)         ||              | (S,G,rpt)   |            |                 |
+------------++--------------+-------------+------------+-----------------+
|NotPruned   || -> P state   | -           | -> NJ state| -               |
|(S,G,rpt)   || Send Prune   |             | Cancel OT  |                 |
|(NP)        || (S,G,rpt);   |             |            |                 |
|            || Cancel OT    |             |            |                 |
+------------++--------------+-------------+------------+-----------------+
```

    Additionally, we have the following transitions within the
    NotPruned(S,G,rpt) state, which are all used for prune override
    behavior.

```
+----------------------------------------------------------------------+
|                     In NotPruned(S,G,rpt) State                      |
+---------+-------------+-------------+-------------+-------------+
|Override | See Prune   | See Join    | See Prune   | RPF'        |
|Timer    | (S,G,rpt) to| (S,G,rpt) to| (S,G) to    | (S,G,rpt) ->|
|expires  | RPF'        | RPF'        | RPF'        | RPF' (*,G)  |
|         | (S,G,rpt)   | (S,G,rpt)   | (S,G,rpt)   |             |
+---------+-------------+-------------+-------------+-------------+
|Send Join| OT = min(OT,| Cancel OT   | OT = min(OT,| OT = min(OT,|
|(S,G,rpt);| t_override) |            | t_override) | t_override) |
|Leave OT |             |             |             |             |
|unset    |             |             |             |             |
+---------+-------------+-------------+-------------+-------------+
```

    Note that the min function in the above state machine considers a
    non-running timer to have an infinite value (e.g., min(not-running,
    t_override) = t_override).

This state machine uses the following macros:

```
  bool RPTJoinDesired(G) {
    return (JoinDesired(*,G) OR JoinDesired(*,*,RP(G)))
  }
```

RPTJoinDesired(G) is true when the router has forwarding state that
would cause it to forward traffic for G using either (*,G) or
(*,*,RP) shared tree state.

```
  bool PruneDesired(S,G,rpt) {
       return ( RPTJoinDesired(G) AND
                ( inherited_olist(S,G,rpt) == NULL
                 OR (SPTbit(S,G)==TRUE
                     AND (RPF'(*,G) != RPF'(S,G)) )))
  }
```

PruneDesired(S,G,rpt) can only be true if RPTJoinDesired(G) is true.
If RPTJoinDesired(G) is true, then PruneDesired(S,G,rpt) is true
either if there are no outgoing interfaces that S would be forwarded
on, or if the router has active (S,G) forwarding state but RPF'(*,G)
!= RPF'(S,G).

The state machine contains the following transition events:

See Join(S,G,rpt) to RPF'(S,G,rpt)
   This event is only relevant in the "Not Pruned" state.

   The router sees a Join(S,G,rpt) from someone else to
   RPF'(S,G,rpt), which is the correct upstream neighbor. If we're
   in "NotPruned" state and the (S,G,rpt) Override Timer is running,
   then this is because we have been triggered to send our own
   Join(S,G,rpt) to RPF'(S,G,rpt). Someone else beat us to it, so
   there's no need to send our own Join.

   The action is to cancel the Override Timer.

See Prune(S,G,rpt) to RPF'(S,G,rpt)
   This event is only relevant in the "NotPruned" state.

   The router sees a Prune(S,G,rpt) from someone else to
   RPF'(S,G,rpt), which is the correct upstream neighbor. If we're
   in the "NotPruned" state, then we want to continue to receive
   traffic from S destined for G, and that traffic is being supplied
   by RPF'(S,G,rpt). Thus, we need to override the Prune.

The action is to set the (S,G,rpt) Override Timer to the
randomized prune-override interval, t_override. However, if the
Override Timer is already running, we only set the timer if doing
so would set it to a lower value. At the end of this interval, if
noone else has sent a Join, then we will do so.

See Prune(S,G) to RPF'(S,G,rpt)
    This event is only relevant in the "NotPruned" state.

    This transition and action are the same as the above transition
    and action, except that the Prune does not have the RPT bit set.
    This transition is necessary to be compatible with routers
    implemented from RFC2362 that don't maintain separate (S,G) and
    (S,G,rpt) state.

The (S,G,rpt) prune Override Timer expires
    This event is only relevant in the "NotPruned" state.

    When the Override Timer expires, we must send a Join(S,G,rpt) to
    RPF'(S,G,rpt) to override the Prune message that caused the timer
    to be running. We only send this if RPF'(S,G,rpt) equals
    RPF'(*,G); if this were not the case, then the Join might be sent
    to a router that does not have (*,G) or (*,*,RP(G)) Join state,
    and so the behavior would not be well defined. If RPF'(S,G,rpt)
    is not the same as RPF'(*,G), then it may stop forwarding S.
    However, if this happens, then the router will send an
    AssertCancel(S,G), which would then cause RPF'(S,G,rpt) to become
    equal to RPF'(*,G) (see below).

RPF'(S,G,rpt) changes to become equal to RPF'(*,G)
    This event is only relevant in the "NotPruned" state.

    RPF'(S,G,rpt) can only be different from RPF'(*,G) if an (S,G)
    Assert has happened, which means that traffic from S is arriving
    on the SPT, and so Prune(S,G,rpt) will have been sent to
    RPF'(*,G). When RPF'(S,G,rpt) changes to become equal to
    RPF'(*,G), we need to trigger a Join(S,G,rpt) to RPF'(*,G) to
    cause that router to start forwarding S again.

    The action is to set the (S,G,rpt) Override Timer to the
    randomized prune-override interval t_override. However, if the
    timer is already running, we only set the timer if doing so would
    set it to a lower value. At the end of this interval, if noone
    else has sent a Join, then we will do so.

PruneDesired(S,G,rpt)->TRUE
    See macro above. This event is relevant in the "NotPruned" and
    "RPTNotJoined(G)" states.

      The router wishes to receive traffic for G, but does not wish to
      receive traffic from S destined for G. This causes the router to
      transition into the Pruned state.

      If the router was previously in NotPruned state, then the action
      is to send a Prune(S,G,rpt) to RPF'(S,G,rpt), and to cancel the
      Override Timer. If the router was previously in RPTNotJoined(G)
      state, then there is no need to trigger an action in this state
      machine because sending a Prune(S,G,rpt) is handled by the rules
      for sending the Join(*,G) or Join(*,*,RP).

   PruneDesired(S,G,rpt)->FALSE
      See macro above. This transition is only relevant in the "Pruned"
      state.

      If the router is in the Pruned(S,G,rpt) state, and
      PruneDesired(S,G,rpt) changes to FALSE, this could be because the
      router no longer has RPTJoinDesired(G) true, or it now wishes to
      receive traffic from S again. If it is the former, then this
      transition should not happen, but instead the
      "RPTJoinDesired(G)->FALSE" transition should happen. Thus, this
      transition should be interpreted as "PruneDesired(S,G,rpt)->FALSE
      AND RPTJoinDesired(G)==TRUE".

      The action is to send a Join(S,G,rpt) to RPF'(S,G,rpt).

   RPTJoinDesired(G)->FALSE
      This event is relevant in the "Pruned" and "NotPruned" states.

      The router no longer wishes to receive any traffic destined for G
      on the RP Tree. This causes a transition to the RPTNotJoined(G)
      state, and the Override Timer is canceled if it was running. Any
      further actions are handled by the appropriate upstream state
      machine for (*,G) or (*,*,RP).

   inherited_olist(S,G,rpt) becomes non-NULL
      This transition is only relevant in the RPTNotJoined(G) state.

      The router has joined the RP tree (handled by the (*,G) or
      (*,*,RP) upstream state machine as appropriate) and wants to
      receive traffic from S. This does not trigger any events in this
      state machine, but causes a transition to the NotPruned(S,G,rpt)
      state.

4.5.10. Background: (*,*,RP) and (S,G,rpt) Interaction

   In Sections 4.5.8 and 4.5.9, the mechanisms for sending periodic and
   triggered (S,G,rpt) messages are described. The astute reader will
   note that periodic Prune(S,G,rpt) messages are only sent in PIM
   Join/Prune messages containing a Join(*,G), whereas it is possible
   for a triggered Prune(S,G,rpt) message to be sent when the router has
   no (*,G) join state. This may seem like a contradiction, but in fact
   it is intentional and is a side effect of not optimizing (*,*,RP)
   behavior.

   We first note that reception of a Join(*,*,RP) by itself does not
   cancel (S,G,rpt) prune state on that interface, whereas receiving a
   Join(*,G) by itself does cancel (S,G,rpt) prune state on that
   interface. Similarly, reception of a Prune(*,G) on an interface with
   (*,*,RP) join state does not by itself prevent forwarding of G using
   the (*,*,RP) state; this is because a Prune(*,G) only serves to
   cancel (*,G) join state. Conceptually (*,*,RP) state functions
   "above" the normal (*,G) and (S,G) mechanisms, and so neither
   Join(*,*,RP) nor Prune(*,*,RP) messages affect any other state.

   The upshot of this is that to prevent forwarding (S,G) on (*,*,RP)
   state, a Prune(S,G,rpt) must be used.

   We also note that for historical reasons there is no Assert(*,*,RP)
   message, so any forwarding contention is resolved using Assert(*,G)
   messages.

   We now need to consider the interaction between (*,*,RP) state and
   (*,G) state. If there is a need for an assert between two upstream
   routers on a LAN, we need to ensure that the correct thing happens
   for all combinations of (*,*,RP) and (*,G) forwarding state. As
   there is no Assert(*,*,RP) message, no router can tell whether the
   assert winner has (*,*,RP) state or (*,G) state. Thus, a downstream
   router has to treat the two the same and send its periodic
   Prune(S,G,rpt) messages to RPF'(*,G).

   To avoid needing to specify all the complex override rules between
   (*,*,RP), (*,G), and (S,G,rpt), we simply require that to prune (S,G)
   off the (*,*,RP) tree, a Join(*,G) must also be sent.

   If a router is receiving on (*,*,RP) state and has not yet had (*,G)
   state instantiated, it may still need to send a triggered
   Join(S,G,rpt) to override a Prune(S,G,rpt) that it sees directed to
   RPF'(*,G) on its upstream interface. Hence, triggered (S,G,rpt)
   messages may be sent when JoinDesired(*,G) is false but
   JoinDesired(*,*,RP) is true.

Finally, we note that there is an unoptimized case when the upstream
router on a LAN already has (*,G) join and (S,G,rpt) prune state
caused by an existing downstream router. If at this time a new
Join(*,*,RP) is sent to the upstream router from a different
downstream router, this will not override the (S,G,rpt) prune state
at the upstream router. The override will not occur until the next
time the original downstream router resends its Prune(S,G,rpt). This
case was not considered worth optimizing, as (*,*,RP) state is
generally very long lived, and so any minor delays in getting traffic
to a new PMBR seem unimportant.

4.6. PIM Assert Messages

Where multiple PIM routers peer over a shared LAN, it is possible for
more than one upstream router to have valid forwarding state for a
packet, which can lead to packet duplication (see Section 3.6). PIM
does not attempt to prevent this from occurring. Instead, it detects
when this has happened and elects a single forwarder amongst the
upstream routers to prevent further duplication. This election is
performed using PIM Assert messages. Assert messages are also
received by downstream routers on the LAN, and these cause subsequent
Join/Prune messages to be sent to the upstream router that won the
Assert.

In general, a PIM Assert message should only be accepted for
processing if it comes from a known PIM neighbor. A PIM router hears
about PIM neighbors through PIM Hello messages. If a router receives
an Assert message from a particular IP source address and it has not
seen a PIM Hello message from that source address, then the Assert
message SHOULD be discarded without further processing. In addition,
if the Hello message from a neighbor was authenticated using the
IPsec Authentication Header (AH) (see Section 6.3), then all Assert
messages from that neighbor MUST also be authenticated using IPsec
AH.

We note that some older PIM implementations incorrectly fail to send
Hello messages on point-to-point interfaces, so we also RECOMMEND
that a configuration option be provided to allow interoperation with
such older routers, but that this configuration option SHOULD NOT be
enabled by default.

4.6.1. (S,G) Assert Message State Machine

The (S,G) Assert state machine for interface I is shown in Figure 10.
There are three states:

NoInfo (NI)
   This router has no (S,G) assert state on interface I.

I am Assert Winner (W)
This router has won an (S,G) assert on interface I. It is now
responsible for forwarding traffic from S destined for G out of
interface I. Irrespective of whether it is the DR for I, while a
router is the assert winner, it is also responsible for forwarding
traffic onto I on behalf of local hosts on I that have made
membership requests that specifically refer to S (and G).

I am Assert Loser (L)
This router has lost an (S,G) assert on interface I. It must not
forward packets from S destined for G onto interface I. If it is
the DR on I, it is no longer responsible for forwarding traffic
onto I to satisfy local hosts with membership requests that
specifically refer to S and G.

In addition, there is also an Assert Timer (AT) that is used to time
out asserts on the assert losers and to resend asserts on the assert
winner.

Figure 10: Per-interface (S,G) Assert State machine in tabular form

```
+---------------------------------------------------------------------+
|                       In NoInfo (NI) State                          |
+---------------+------------------+-----------------+----------------+
| Receive       | Receive Assert   | Data arrives    | Receive        |
| Inferior      | with RPTbit      | from S to G on  | Acceptable     |
| Assert with   | set and          | I and           | Assert with    |
| RPTbit clear  | CouldAssert      | CouldAssert     | RPTbit clear   |
| and           | (S,G,I)          | (S,G,I)         | and AssTrDes   |
| CouldAssert   |                  |                 | (S,G,I)        |
| (S,G,I)       |                  |                 |                |
+---------------+------------------+-----------------+----------------+
| -> W state    | -> W state       | -> W state      | -> L state     |
| [Actions A1]  | [Actions A1]     | [Actions A1]    | [Actions A6]   |
+---------------+------------------+-----------------+----------------+


+---------------------------------------------------------------------+
|                 In I Am Assert Winner (W) State                     |
+---------------+------------------+-----------------+----------------+
| Assert Timer  | Receive          | Receive         | CouldAssert    |
| Expires       | Inferior         | Preferred       | (S,G,I) ->     |
|               | Assert           | Assert          | FALSE          |
+---------------+------------------+-----------------+----------------+
| -> W state    | -> W state       | -> L state      | -> NI state    |
| [Actions A3]  | [Actions A3]     | [Actions A2]    | [Actions A4]   |
+---------------+------------------+-----------------+----------------+
```

```
+---------------------------------------------------------------------+
|                  In I Am Assert Loser (L) State                     |
+-------------+-------------+-------------+-------------+-------------+
|Receive      |Receive      |Receive      |Assert Timer |Current      |
|Preferred    |Acceptable   |Inferior     |Expires      |Winner's     |
|Assert       |Assert with  |Assert or    |             |GenID        |
|             |RPTbit clear |Assert       |             |Changes or   |
|             |from Current |Cancel from  |             |NLT Expires  |
|             |Winner       |Current      |             |             |
|             |             |Winner       |             |             |
+-------------+-------------+-------------+-------------+-------------+
|-> L state   |-> L state   |-> NI state  |-> NI state  |-> NI state  |
|[Actions A2] |[Actions A2] |[Actions A5] |[Actions A5] |[Actions A5] |
+-------------+-------------+-------------+-------------+-------------+


+----------------------------------------------------------------------+
|                  In I Am Assert Loser (L) State                      |
+---------------+----------------+----------------+----------------+
| AssTrDes      | my_metric ->   | RPF_interface  | Receive        |
| (S,G,I) ->    | better than    | (S) stops      | Join(S,G) on   |
| FALSE         | winner's       | being I        | interface I    |
|               | metric         |                |                |
+---------------+----------------+----------------+----------------+
| -> NI state   | -> NI state    | -> NI state    | -> NI State    |
| [Actions A5]  | [Actions A5]   | [Actions A5]   | [Actions A5]   |
+---------------+----------------+----------------+----------------+
```

   Note that for reasons of compactness, "AssTrDes(S,G,I)" is used in
   the state machine table to refer to AssertTrackingDesired(S,G,I).

   Terminology:

      A "preferred assert" is one with a better metric than the current
      winner.

      An "acceptable assert" is one that has a better metric than
      my_assert_metric(S,G,I). An assert is never considered acceptable
      if its metric is infinite.

      An "inferior assert" is one with a worse metric than
      my_assert_metric(S,G,I). An assert is never considered inferior
      if my_assert_metric(S,G,I) is infinite.

    The state machine uses the following macros:

CouldAssert(S,G,I) =
     SPTbit(S,G)==TRUE
     AND (RPF_interface(S) != I)
     AND (I in ((joins(*,*,RP(G)) (+) joins(*,G) (-) prunes(S,G,rpt))
                 (+) (pim_include(*,G) (-) pim_exclude(S,G))
                 (-) lost_assert(*,G)
                 (+) joins(S,G) (+) pim_include(S,G)))

    CouldAssert(S,G,I) is true for downstream interfaces that would be in
    the inherited_olist(S,G) if (S,G) assert information was not taken
    into account.

    AssertTrackingDesired(S,G,I) =
        (I in ((joins(*,*,RP(G)) (+) joins(*,G) (-) prunes(S,G,rpt))
                (+) (pim_include(*,G) (-) pim_exclude(S,G))
                (-) lost_assert(*,G)
                (+) joins(S,G)))
        OR (local_receiver_include(S,G,I) == TRUE
            AND (I_am_DR(I) OR (AssertWinner(S,G,I) == me)))
        OR ((RPF_interface(S) == I) AND (JoinDesired(S,G) == TRUE))
        OR ((RPF_interface(RP(G)) == I) AND (JoinDesired(*,G) == TRUE)
            AND (SPTbit(S,G) == FALSE))

    AssertTrackingDesired(S,G,I) is true on any interface in which an
    (S,G) assert might affect our behavior.

    The first three lines of AssertTrackingDesired account for (*,G) join
    and local membership information received on I that might cause the
    router to be interested in asserts on I.

    The 4th line accounts for (S,G) join information received on I that
    might cause the router to be interested in asserts on I.

    The 5th and 6th lines account for (S,G) local membership information
    on I. Note that we can't use the pim_include(S,G) macro since it
    uses lost_assert(S,G,I) and would result in the router forgetting
    that it lost an assert if the only reason it was interested was local
    membership. The AssertWinner(S,G,I) check forces an assert winner to
    keep on being responsible for forwarding as long as local receivers
    are present. Removing this check would make the assert winner give
    up forwarding as soon as the information that originally caused it to
    forward went away, and the task of forwarding for local receivers
    would revert back to the DR.

The last three lines account for the fact that a router must keep
track of assert information on upstream interfaces in order to send
joins and prunes to the proper neighbor.

Transitions from NoInfo State

When in NoInfo state, the following events may trigger transitions:

   Receive Inferior Assert with RPTbit cleared AND
        CouldAssert(S,G,I)==TRUE
        An assert is received for (S,G) with the RPT bit cleared that
        is inferior to our own assert metric. The RPT bit cleared
        indicates that the sender of the assert had (S,G) forwarding
        state on this interface. If the assert is inferior to our
        metric, then we must also have (S,G) forwarding state (i.e.,
        CouldAssert(S,G,I)==TRUE) as (S,G) asserts beat (*,G) asserts,
        and so we should be the assert winner. We transition to the
        "I am Assert Winner" state and perform Actions A1 (below).

   Receive Assert with RPTbit set AND CouldAssert(S,G,I)==TRUE
        An assert is received for (S,G) on I with the RPT bit set
        (it's a (*,G) assert). CouldAssert(S,G,I) is TRUE only if we
        have (S,G) forwarding state on this interface, so we should be
        the assert winner. We transition to the "I am Assert Winner"
        state and perform Actions A1 (below).

   An (S,G) data packet arrives on interface I, AND
        CouldAssert(S,G,I)==TRUE
        An (S,G) data packet arrived on an downstream interface that
        is in our (S,G) outgoing interface list. We optimistically
        assume that we will be the assert winner for this (S,G), and
        so we transition to the "I am Assert Winner" state and perform
        Actions A1 (below), which will initiate the assert negotiation
        for (S,G).

   Receive Acceptable Assert with RPT bit clear AND
        AssertTrackingDesired(S,G,I)==TRUE
        We're interested in (S,G) Asserts, either because I is a
        downstream interface for which we have (S,G) or (*,G)
        forwarding state, or because I is the upstream interface for S
        and we have (S,G) forwarding state. The received assert has a
        better metric than our own, so we do not win the Assert. We
        transition to "I am Assert Loser" and perform Actions A6
        (below).

   Transitions from "I am Assert Winner" State

   When in "I am Assert Winner" state, the following events trigger
   transitions:

     Assert Timer Expires
           The (S,G) Assert Timer expires. As we're in the Winner state,
           we must still have (S,G) forwarding state that is actively
           being kept alive. We resend the (S,G) Assert and restart the
           Assert Timer (Actions A3 below). Note that the assert
           winner's Assert Timer is engineered to expire shortly before
           timers on assert losers; this prevents unnecessary thrashing
           of the forwarder and periodic flooding of duplicate packets.

     Receive Inferior Assert
           We receive an (S,G) assert or (*,G) assert mentioning S that
           has a worse metric than our own. Whoever sent the assert is
           in error, and so we resend an (S,G) Assert and restart the
           Assert Timer (Actions A3 below).

     Receive Preferred Assert
           We receive an (S,G) assert that has a better metric than our
           own. We transition to "I am Assert Loser" state and perform
           Actions A2 (below). Note that this may affect the value of
           JoinDesired(S,G) and PruneDesired(S,G,rpt), which could cause
           transitions in the upstream (S,G) or (S,G,rpt) state machines.

     CouldAssert(S,G,I) -> FALSE
           Our (S,G) forwarding state or RPF interface changed so as to
           make CouldAssert(S,G,I) become false. We can no longer
           perform the actions of the assert winner, and so we transition
           to NoInfo state and perform Actions A4 (below). This includes
           sending a "canceling assert" with an infinite metric.

   Transitions from "I am Assert Loser" State

   When in "I am Assert Loser" state, the following transitions can
   occur:

     Receive Preferred Assert
           We receive an assert that is better than that of the current
           assert winner. We stay in Loser state and perform Actions A2
           below.

Receive Acceptable Assert with RPTbit clear from Current Winner
     We receive an assert from the current assert winner that is
     better than our own metric for this (S,G) (although the metric
     may be worse than the winner's previous metric). We stay in
     Loser state and perform Actions A2 below.

Receive Inferior Assert or Assert Cancel from Current Winner
     We receive an assert from the current assert winner that is
     worse than our own metric for this group (typically, because
     the winner's metric became worse or because it is an assert
     cancel). We transition to NoInfo state, deleting the (S,G)
     assert information and allowing the normal PIM Join/Prune
     mechanisms to operate. Usually, we will eventually re-assert
     and win when data packets from S have started flowing again.

Assert Timer Expires
     The (S,G) Assert Timer expires. We transition to NoInfo
     state, deleting the (S,G) assert information (Actions A5
     below).

Current Winner's GenID Changes or NLT Expires
     The Neighbor Liveness Timer associated with the current winner
     expires or we receive a Hello message from the current winner
     reporting a different GenID from the one it previously
     reported. This indicates that the current winner's interface
     or router has gone down (and may have come back up), and so we
     must assume it no longer knows it was the winner. We
     transition to the NoInfo state, deleting this (S,G) assert
     information (Actions A5 below).

AssertTrackingDesired(S,G,I)->FALSE
     AssertTrackingDesired(S,G,I) becomes FALSE. Our forwarding
     state has changed so that (S,G) Asserts on interface I are no
     longer of interest to us. We transition to the NoInfo state,
     deleting the (S,G) assert information.

My metric becomes better than the assert winner's metric
     my_assert_metric(S,G,I) has changed so that now my assert
     metric for (S,G) is better than the metric we have stored for
     current assert winner. This might happen when the underlying
     routing metric changes, or when CouldAssert(S,G,I) becomes
     true; for example, when SPTbit(S,G) becomes true. We
     transition to NoInfo state, delete this (S,G) assert state
     (Actions A5 below), and allow the normal PIM Join/Prune
     mechanisms to operate. Usually, we will eventually re-assert
     and win when data packets from S have started flowing again.

    RPF_interface(S) stops being interface I
         Interface I used to be the RPF interface for S, and now it is
         not. We transition to NoInfo state, deleting this (S,G)
         assert state (Actions A5 below).

    Receive Join(S,G) on Interface I
         We receive a Join(S,G) that has the Upstream Neighbor Address
         field set to my primary IP address on interface I. The action
         is to transition to NoInfo state, delete this (S,G) assert
         state (Actions A5 below), and allow the normal PIM Join/Prune
         mechanisms to operate. If whoever sent the Join was in error,
         then the normal assert mechanism will eventually re-apply, and
         we will lose the assert again. However, whoever sent the
         assert may know that the previous assert winner has died, and
         so we may end up being the new forwarder.

  (S,G) Assert State machine Actions

    A1: Send Assert(S,G).
         Set Assert Timer to (Assert_Time - Assert_Override_Interval).
         Store self as AssertWinner(S,G,I).
         Store spt_assert_metric(S,I) as AssertWinnerMetric(S,G,I).

    A2: Store new assert winner as AssertWinner(S,G,I) and assert
         winner metric as AssertWinnerMetric(S,G,I).
         Set Assert Timer to Assert_Time.

    A3: Send Assert(S,G).
         Set Assert Timer to (Assert_Time - Assert_Override_Interval).

    A4: Send AssertCancel(S,G).
         Delete assert info (AssertWinner(S,G,I) and
         AssertWinnerMetric(S,G,I) will then return their default
         values).

    A5: Delete assert info (AssertWinner(S,G,I) and
         AssertWinnerMetric(S,G,I) will then return their default
         values).

    A6: Store new assert winner as AssertWinner(S,G,I) and assert
         winner metric as AssertWinnerMetric(S,G,I).
         Set Assert Timer to Assert_Time.
         If (I is RPF_interface(S)) AND (UpstreamJPState(S,G) == true)
         set SPTbit(S,G) to TRUE.

  Note that some of these actions may cause the value of
  JoinDesired(S,G), PruneDesired(S,G,rpt), or RPF'(S,G) to change,
  which could cause further transitions in other state machines.

4.6.2. (*,G) Assert Message State Machine

   The (*,G) Assert state machine for interface I is shown in Figure 11.
   There are three states:

   NoInfo (NI)
      This router has no (*,G) assert state on interface I.

   I am Assert Winner (W)
      This router has won an (*,G) assert on interface I. It is now
      responsible for forwarding traffic destined for G onto interface I
      with the exception of traffic for which it has (S,G) "I am Assert
      Loser" state. Irrespective of whether it is the DR for I, it is
      also responsible for handling the membership requests for G from
      local hosts on I.

   I am Assert Loser (L)
      This router has lost an (*,G) assert on interface I. It must not
      forward packets for G onto interface I with the exception of
      traffic from sources for which is has (S,G) "I am Assert Winner"
      state. If it is the DR, it is no longer responsible for handling
      the membership requests for group G from local hosts on I.

   In addition, there is also an Assert Timer (AT) that is used to time
   out asserts on the assert losers and to resend asserts on the assert
   winner.

   When an Assert message is received with a source address other than
   zero, a PIM implementation must first match it against the possible
   events in the (S,G) assert state machine and process any transitions
   and actions, before considering whether the Assert message matches
   against the (*,G) assert state machine.

   It is important to note that NO TRANSITION CAN OCCUR in the (*,G)
   state machine as a result of receiving an Assert message unless the
   (S,G) assert state machine for the relevant S and G is in the
   "NoInfo" state after the (S,G) state machine has processed the
   message. Also, NO TRANSITION CAN OCCUR in the (*,G) state machine as
   a result of receiving an assert message if that message triggers any
   change of state in the (S,G) state machine. Obviously, when the
   source address in the received message is set to zero, an (S,G) state
   machine for the S and G does not exist and can be assumed to be in
   the "NoInfo" state.

For example, if both the (S,G) and (*,G) assert state machines are in
the NoInfo state when an Assert message arrives, and the message
causes the (S,G) state machine to transition to either "W" or "L"
state, then the assert will not be processed by the (*,G) assert
state machine.

Another example: if the (S,G) assert state machine is in "L" state
when an assert message is received, and the assert metric in the
message is worse than my_assert_metric(S,G,I), then the (S,G) assert
state machine will transition to NoInfo state. In such a case, if
the (*,G) assert state machine were in NoInfo state, it might appear
that it would transition to "W" state, but this is not the case
because this message already triggered a transition in the (S,G)
assert state machine.

Figure 11: Per-interface (*,G) Assert State machine in tabular form

In NoInfo (NI) State		
Receive Inferior Assert with RPTbit set and CouldAssert(*,G,I)	Data arrives for G on I and CouldAssert (*,G,I)	Receive Acceptable Assert with RPTbit set and AssTrDes (*,G,I)
-> W state [Actions A1]	-> W state [Actions A1]	-> L state [Actions A2]

In I Am Assert Winner (W) State			
Assert Timer Expires	Receive Inferior Assert	Receive Preferred Assert	CouldAssert (*,G,I) -> FALSE
-> W state [Actions A3]	-> W state [Actions A3]	-> L state [Actions A2]	-> NI state [Actions A4]

```
+---------------------------------------------------------------------+
|                 In I Am Assert Loser (L) State                      |
+-------------+-------------+-------------+-------------+-------------+
|Receive      |Receive      |Receive      |Assert Timer |Current      |
|Preferred    |Acceptable   |Inferior     |Expires      |Winner's     |
|Assert with  |Assert from  |Assert or    |             |GenID        |
|RPTbit set   |Current      |Assert       |             |Changes or   |
|             |Winner with  |Cancel from  |             |NLT Expires  |
|             |RPTbit set   |Current      |             |             |
|             |             |Winner       |             |             |
+-------------+-------------+-------------+-------------+-------------+
|-> L state   |-> L state   |-> NI state  |-> NI state  |-> NI state  |
|[Actions A2] |[Actions A2] |[Actions A5] |[Actions A5] |[Actions A5] |
+-------------+-------------+-------------+-------------+-------------+


+---------------------------------------------------------------------+
|                 In I Am Assert Loser (L) State                      |
+---------------+---------------+---------------+-----------------+
| AssTrDes      | my_metric ->  | RPF_interface | Receive         |
| (*,G,I) ->    | better than   | (RP(G)) stops | Join(*,G) or    |
| FALSE         | Winner's      | being I       | Join            |
|               | metric        |               | (*,*,RP(G)) on  |
|               |               |               | Interface I     |
+---------------+---------------+---------------+-----------------+
| -> NI state   | -> NI state   | -> NI state   | -> NI State     |
| [Actions A5]  | [Actions A5]  | [Actions A5]  | [Actions A5]    |
+---------------+---------------+---------------+-----------------+
```

   The state machine uses the following macros:

   CouldAssert(*,G,I) =
        (I in (joins(*,*,RP(G)) (+) joins(*,G)
                 (+) pim_include(*,G)))
        AND (RPF_interface(RP(G)) != I)

   CouldAssert(*,G,I) is true on downstream interfaces for which we have
   (*,*,RP(G)) or (*,G) join state, or local members that requested any
   traffic destined for G.

   AssertTrackingDesired(*,G,I) =
        CouldAssert(*,G,I)
        OR (local_receiver_include(*,G,I)==TRUE
             AND (I_am_DR(I) OR AssertWinner(*,G,I) == me))
        OR (RPF_interface(RP(G)) == I AND RPTJoinDesired(G))

   AssertTrackingDesired(*,G,I) is true on any interface on which an
   (*,G) assert might affect our behavior.

Note that for reasons of compactness, "AssTrDes(*,G,I)" is used in
the state machine table to refer to AssertTrackingDesired(*,G,I).

Terminology:

   A "preferred assert" is one with a better metric than the current
   winner.

   An "acceptable assert" is one that has a better metric than
   my_assert_metric(*,G,I). An assert is never considered acceptable
   if its metric is infinite.

   An "inferior assert" is one with a worse metric than
   my_assert_metric(*,G,I). An assert is never considered inferior
   if my_assert_metric(*,G,I) is infinite.

Transitions from NoInfo State

When in NoInfo state, the following events trigger transitions, but
only if the (S,G) assert state machine is in NoInfo state before and
after consideration of the received message:

  Receive Inferior Assert with RPTbit set AND
       CouldAssert(*,G,I)==TRUE
       An Inferior (*,G) assert is received for G on Interface I. If
       CouldAssert(*,G,I) is TRUE, then I is our downstream
       interface, and we have (*,G) forwarding state on this
       interface, so we should be the assert winner. We transition
       to the "I am Assert Winner" state and perform Actions A1
       (below).

  A data packet destined for G arrives on interface I, AND
       CouldAssert(*,G,I)==TRUE
       A data packet destined for G arrived on a downstream interface
       that is in our (*,G) outgoing interface list. We therefore
       believe we should be the forwarder for this (*,G), and so we
       transition to the "I am Assert Winner" state and perform
       Actions A1 (below).

  Receive Acceptable Assert with RPT bit set AND
       AssertTrackingDesired(*,G,I)==TRUE
       We're interested in (*,G) Asserts, either because I is a
       downstream interface for which we have (*,G) forwarding state,
       or because I is the upstream interface for RP(G) and we have
       (*,G) forwarding state. We get a (*,G) Assert that has a
       better metric than our own, so we do not win the Assert. We
       transition to "I am Assert Loser" and perform Actions A2
       (below).

Transitions from "I am Assert Winner" State

When in "I am Assert Winner" state, the following events trigger
transitions, but only if the (S,G) assert state machine is in NoInfo
state before and after consideration of the received message:

   Receive Inferior Assert
        We receive a (*,G) assert that has a worse metric than our
        own. Whoever sent the assert has lost, and so we resend a
        (*,G) Assert and restart the Assert Timer (Actions A3 below).

   Receive Preferred Assert
        We receive a (*,G) assert that has a better metric than our
        own. We transition to "I am Assert Loser" state and perform
        Actions A2 (below).

When in "I am Assert Winner" state, the following events trigger
transitions:

   Assert Timer Expires
        The (*,G) Assert Timer expires. As we're in the Winner state,
        then we must still have (*,G) forwarding state that is
        actively being kept alive. To prevent unnecessary thrashing
        of the forwarder and periodic flooding of duplicate packets,
        we resend the (*,G) Assert and restart the Assert Timer
        (Actions A3 below).

   CouldAssert(*,G,I) -> FALSE
        Our (*,G) forwarding state or RPF interface changed so as to
        make CouldAssert(*,G,I) become false. We can no longer
        perform the actions of the assert winner, and so we transition
        to NoInfo state and perform Actions A4 (below).

Transitions from "I am Assert Loser" State

When in "I am Assert Loser" state, the following events trigger
transitions, but only if the (S,G) assert state machine is in NoInfo
state before and after consideration of the received message:

   Receive Preferred Assert with RPTbit set
        We receive a (*,G) assert that is better than that of the
        current assert winner. We stay in Loser state and perform
        Actions A2 below.

    Receive Acceptable Assert from Current Winner with RPTbit set
         We receive a (*,G) assert from the current assert winner that
         is better than our own metric for this group (although the
         metric may be worse than the winner's previous metric). We
         stay in Loser state and perform Actions A2 below.

    Receive Inferior Assert or Assert Cancel from Current Winner
         We receive an assert from the current assert winner that is
         worse than our own metric for this group (typically because
         the winner's metric became worse or is now an assert cancel).
         We transition to NoInfo state, delete this (*,G) assert state
         (Actions A5), and allow the normal PIM Join/Prune mechanisms
         to operate. Usually, we will eventually re-assert and win
         when data packets for G have started flowing again.

   When in "I am Assert Loser" state, the following events trigger
   transitions:

    Assert Timer Expires
         The (*,G) Assert Timer expires. We transition to NoInfo state
         and delete this (*,G) assert info (Actions A5).

    Current Winner's GenID Changes or NLT Expires
         The Neighbor Liveness Timer associated with the current winner
         expires or we receive a Hello message from the current winner
         reporting a different GenID from the one it previously
         reported. This indicates that the current winner's interface
         or router has gone down (and may have come back up), and so we
         must assume it no longer knows it was the winner. We
         transition to the NoInfo state, deleting the (*,G) assert
         information (Actions A5).

    AssertTrackingDesired(*,G,I)->FALSE
         AssertTrackingDesired(*,G,I) becomes FALSE. Our forwarding
         state has changed so that (*,G) Asserts on interface I are no
         longer of interest to us. We transition to NoInfo state and
         delete this (*,G) assert info (Actions A5).

    My metric becomes better than the assert winner's metric
         My routing metric, rpt_assert_metric(G,I), has changed so that
         now my assert metric for (*,G) is better than the metric we
         have stored for current assert winner. We transition to
         NoInfo state, delete this (*,G) assert state (Actions A5), and
         allow the normal PIM Join/Prune mechanisms to operate.
         Usually, we will eventually re-assert and win when data
         packets for G have started flowing again.

          RPF_interface(RP(G)) stops being interface I
               Interface I used to be the RPF interface for RP(G), and now it
               is not. We transition to NoInfo state and delete this (*,G)
               assert state (Actions A5).

     Receive Join(*,G) or Join(*,*,RP(G)) on interface I
               We receive a Join(*,G) or a Join(*,*,RP(G)) that has the
               Upstream Neighbor Address field set to my primary IP address
               on interface I. The action is to transition to NoInfo state,
               delete this (*,G) assert state (Actions A5), and allow the
               normal PIM Join/Prune mechanisms to operate. If whoever sent
               the Join was in error, then the normal assert mechanism will
               eventually re-apply, and we will lose the assert again.
               However, whoever sent the assert may know that the previous
               assert winner has died, so we may end up being the new
               forwarder.

     (*,G) Assert State machine Actions

       A1: Send Assert(*,G).
            Set Assert Timer to (Assert_Time - Assert_Override_Interval).
            Store self as AssertWinner(*,G,I).
            Store rpt_assert_metric(G,I) as AssertWinnerMetric(*,G,I).

       A2: Store new assert winner as AssertWinner(*,G,I) and assert
            winner metric as AssertWinnerMetric(*,G,I).
            Set Assert Timer to Assert_Time.

       A3: Send Assert(*,G)
            Set Assert Timer to (Assert_Time - Assert_Override_Interval).

       A4: Send AssertCancel(*,G).
            Delete assert info (AssertWinner(*,G,I) and
            AssertWinnerMetric(*,G,I) will then return their default
            values).

       A5: Delete assert info (AssertWinner(*,G,I) and
            AssertWinnerMetric(*,G,I) will then return their default
            values).

   Note that some of these actions may cause the value of
   JoinDesired(*,G) or RPF'(*,G)) to change, which could cause further
   transitions in other state machines.

4.6.3. Assert Metrics

   Assert metrics are defined as:

      struct assert_metric {
        rpt_bit_flag;
        metric_preference;
        route_metric;
        ip_address;
      };

   When comparing assert_metrics, the rpt_bit_flag, metric_preference,
   and route_metric field are compared in order, where the first lower
   value wins. If all fields are equal, the primary IP address of the
   router that sourced the Assert message is used as a tie-breaker, with
   the highest IP address winning.

   An assert metric for (S,G) to include in (or compare against) an
   Assert message sent on interface I should be computed using the
   following pseudocode:

      assert_metric
      my_assert_metric(S,G,I) {
          if(CouldAssert(S,G,I) == TRUE) {
              return spt_assert_metric(S,I)
          } else if(CouldAssert(*,G,I) == TRUE) {
              return rpt_assert_metric(G,I)
          } else {
              return infinite_assert_metric()
          }
      }

   spt_assert_metric(S,I) gives the assert metric we use if we're
   sending an assert based on active (S,G) forwarding state:

      assert_metric
      spt_assert_metric(S,I) {
          return {0,MRIB.pref(S),MRIB.metric(S),my_ip_address(I)}
      }

   rpt_assert_metric(G,I) gives the assert metric we use if we're
   sending an assert based only on (*,G) forwarding state:

      assert_metric
      rpt_assert_metric(G,I) {
          return {1,MRIB.pref(RP(G)),MRIB.metric(RP(G)),my_ip_address(I)}
      }

MRIB.pref(X) and MRIB.metric(X) are the routing preference and
routing metrics associated with the route to a particular (unicast)
destination X, as determined by the MRIB. my_ip_address(I) is simply
the router's primary IP address that is associated with the local
interface I.

infinite_assert_metric() gives the assert metric we need to send an
assert but don't match either (S,G) or (*,G) forwarding state:

```
assert_metric
infinite_assert_metric() {
     return {1,infinity,infinity,0}
}
```

4.6.4. AssertCancel Messages

An AssertCancel message is simply an RPT Assert message but with
infinite metric. It is sent by the assert winner when it deletes the
forwarding state that had caused the assert to occur. Other routers
will see this metric, and it will cause any other router that has
forwarding state to send its own assert, and to take over forwarding.

An AssertCancel(S,G) is an infinite metric assert with the RPT bit
set that names S as the source.

An AssertCancel(*,G) is an infinite metric assert with the RPT bit
set and the source set to zero.

AssertCancel messages are simply an optimization. The original
Assert timeout mechanism will allow a subnet to eventually become
consistent; the AssertCancel mechanism simply causes faster
convergence. No special processing is required for an AssertCancel
message, since it is simply an Assert message from the current
winner.

4.6.5. Assert State Macros

   The macros lost_assert(S,G,rpt,I), lost_assert(S,G,I), and
   lost_assert(*,G,I) are used in the olist computations of Section 4.1,
   and are defined as:

```
     bool lost_assert(S,G,rpt,I) {
       if ( RPF_interface(RP(G)) == I  OR
            ( RPF_interface(S) == I AND SPTbit(S,G) == TRUE ) ) {
         return FALSE
       } else {
         return ( AssertWinner(S,G,I) != NULL AND
                  AssertWinner(S,G,I) != me )
       }
     }


     bool lost_assert(S,G,I) {
       if ( RPF_interface(S) == I ) {
         return FALSE
       } else {
         return ( AssertWinner(S,G,I) != NULL AND
                  AssertWinner(S,G,I) != me  AND
                  (AssertWinnerMetric(S,G,I) is better
                     than spt_assert_metric(S,I) )
       }
     }
```

   Note: the term "AssertWinnerMetric(S,G,I) is better than
   spt_assert_metric(S,I)" is required to correctly handle the
   transition phase when a router has (S,G) join state, but has not yet
   set the SPT bit. In this case, it needs to ignore the assert state
   if it will win the assert once the SPTbit is set.

```
     bool lost_assert(*,G,I) {
       if ( RPF_interface(RP(G)) == I ) {
         return FALSE
       } else {
         return ( AssertWinner(*,G,I) != NULL AND
                  AssertWinner(*,G,I) != me )
       }
     }
```

   AssertWinner(S,G,I) is the IP source address of the Assert(S,G)
   packet that won an Assert.

   AssertWinner(*,G,I) is the IP source address of the Assert(*,G)
   packet that won an Assert.

AssertWinnerMetric(S,G,I) is the Assert metric of the Assert(S,G)
packet that won an Assert.

AssertWinnerMetric(*,G,I) is the Assert metric of the Assert(*,G)
packet that won an Assert.

AssertWinner(S,G,I) defaults to NULL and AssertWinnerMetric(S,G,I)
defaults to Infinity when in the NoInfo state.

Summary of Assert Rules and Rationale

This section summarizes the key rules for sending and reacting to
asserts and the rationale for these rules. This section is not
intended to be and should not be treated as a definitive
specification of protocol behavior. The state machines and
pseudocode should be consulted for that purpose. Rather, this
section is intended to document important aspects of the Assert
protocol behavior and to provide information that may prove helpful
to the reader in understanding and implementing this part of the
protocol.

1. Behavior: Downstream neighbors send Join(*,G) and Join(S,G)
    periodic messages to the appropriate RPF' neighbor, i.e., the RPF
    neighbor as modified by the assert process. They are not always
    sent to the RPF neighbor as indicated by the MRIB. Normal
    suppression and override rules apply.

    Rationale: By sending the periodic and triggered Join messages to
    the RPF' neighbor instead of to the RPF neighbor, the downstream
    router avoids re-triggering the Assert process with every Join.
    A side effect of sending Joins to the Assert winner is that
    traffic will not switch back to the "normal" RPF neighbor until
    the Assert times out. This will not happen until data stops
    flowing, if item 8, below, is implemented.

2. Behavior: The assert winner for (*,G) acts as the local DR for
    (*,G) on behalf of IGMP/MLD members.

    Rationale: This is required to allow a single router to merge PIM
    and IGMP/MLD joins and leaves. Without this, overrides don't
    work.

3. Behavior: The assert winner for (S,G) acts as the local DR for
    (S,G) on behalf of IGMPv3 members.

    Rationale: Same rationale as for item 2.

4. Behavior: (S,G) and (*,G) prune overrides are sent to the RPF'
    neighbor and not to the regular RPF neighbor.

    Rationale: Same rationale as for item 1.

5. Behavior: An (S,G,rpt) prune override is not sent (at all) if
    RPF'(S,G,rpt) != RPF'(*,G).

    Rationale: This avoids keeping state alive on the (S,G) tree when
    only (*,G) downstream members are left. Also, it avoids sending
    (S,G,rpt) joins to a router that is not on the (*,G) tree. This
    behavior might be confusing although this specification does
    indicate that such a join should be dropped.

6. Behavior: An assert loser that receives a Join(S,G) with an
    Upstream Neighbor Address that is its primary IP address on that
    interface cancels the (S,G) Assert Timer.

    Rationale: This is necessary in order to have rapid convergence
    in the event that the downstream router that initially sent a
    join to the prior Assert winner has undergone a topology change.

7. Behavior: An assert loser that receives a Join(*,G) or a
    Join(*,*,RP(G)) with an Upstream Neighbor Address that is its
    primary IP address on that interface cancels the (*,G) Assert
    Timer and all (S,G) assert timers that do not have corresponding
    Prune(S,G,rpt) messages in the compound Join/Prune message.

    Rationale: Same rationale as for item 6.

8. Behavior: An assert winner for (*,G) or (S,G) sends a canceling
    assert when it is about to stop forwarding on a (*,G) or an (S,G)
    entry. This behavior does not apply to (S,G,rpt).

    Rationale: This allows switching back to the shared tree after
    the last SPT router on the LAN leaves. Doing this prevents
    downstream routers on the shared tree from keeping SPT state
    alive.

9. Behavior: Resend the assert messages before timing out an assert.
    (This behavior is optional.)

    Rationale: This prevents the periodic duplicates that would
    otherwise occur each time that an assert times out and is then
    re-established.

10. Behavior: When RPF'(S,G,rpt) changes to be the same as RPF'(*,G)
    we need to trigger a Join(S,G,rpt) to RPF'(*,G).

         Rationale: This allows switching back to the RPT after the last
         SPT member leaves.

4.7. PIM Bootstrap and RP Discovery

   For correct operation, every PIM router within a PIM domain must be
   able to map a particular multicast group address to the same RP. If
   this is not the case, then black holes may appear, where some
   receivers in the domain cannot receive some groups. A domain in this
   context is a contiguous set of routers that all implement PIM and are
   configured to operate within a common boundary.

   A notable exception to this is where a PIM domain is broken up into
   multiple administrative scope regions; these are regions where a
   border has been configured so that a range of multicast groups will
   not be forwarded across that border. For more information on
   Administratively Scoped IP Multicast, see RFC 2365. The modified
   criteria for admin-scoped regions are that the region is convex with
   respect to forwarding based on the MRIB, and that all PIM routers
   within the scope region map scoped groups to the same RP within that
   region.

   This specification does not mandate the use of a single mechanism to
   provide routers with the information to perform the group-to-RP
   mapping. Currently four mechanisms are possible, and all four have
   associated problems:

   Static Configuration
         A PIM router MUST support the static configuration of group-to-
         RP mappings. Such a mechanism is not robust to failures, but
         does at least provide a basic interoperability mechanism.

   Embedded-RP
         Embedded-RP defines an address allocation policy in which the
         address of the Rendezvous Point (RP) is encoded in an IPv6
         multicast group address [17].

   Cisco's Auto-RP
         Auto-RP uses a PIM Dense-Mode multicast group to announce
         group-to-RP mappings from a central location. This mechanism is
         not useful if PIM Dense-Mode is not being run in parallel with
         PIM Sparse-Mode, and was only intended for use with PIM Sparse-
         Mode Version 1. No standard specification currently exists.

   BootStrap Router (BSR)
         RFC 2362 specifies a bootstrap mechanism based on the automatic
         election of a bootstrap router (BSR). Any router in the domain
         that is configured to be a possible RP reports its candidacy to

the BSR, and then a domain-wide flooding mechanism distributes
the BSR's chosen set of RPs throughout the domain. As specified
in RFC 2362, BSR is flawed in its handling of admin-scoped
regions that are smaller than a PIM domain, but the mechanism
does work for global-scoped groups.

As far as PIM-SM is concerned, the only important requirement is that
all routers in the domain (or admin scope zone for scoped regions)
receive the same set of group-range-to-RP mappings. This may be
achieved through the use of any of these mechanisms, or through
alternative mechanisms not currently specified.

It must be operationally ensured that any RP address configured,
learned, or advertised is reachable from all routers in the PIM
domain.

4.7.1. Group-to-RP Mapping

Using one of the mechanisms described above, a PIM router receives
one or more possible group-range-to-RP mappings. Each mapping
specifies a range of multicast groups (expressed as a group and mask)
and the RP to which such groups should be mapped. Each mapping may
also have an associated priority. It is possible to receive multiple
mappings, all of which might match the same multicast group; this is
the common case with BSR. The algorithm for performing the group-
to-RP mapping is as follows:

1. Perform longest match on group-range to obtain a list of RPs.

2. From this list of matching RPs, find the one with highest
    priority. Eliminate any RPs from the list that have lower
    priorities.

3. If only one RP remains in the list, use that RP.

4. If multiple RPs are in the list, use the PIM hash function to
    choose one.

Thus, if two or more group-range-to-RP mappings cover a particular
group, the one with the longest mask is the mapping to use. If the
mappings have the same mask length, then the one with the highest
priority is chosen. If there is more than one matching entry with
the same longest mask and the priorities are identical, then a hash
function (see Section 4.7.2) is applied to choose the RP.

This algorithm is invoked by a DR when it needs to determine an RP
for a given group, e.g., upon reception of a packet or IGMP/MLD
membership indication for a group for which the DR does not know the

RP. It is invoked by any router that has (*,*,RP) state when a
packet is received for which there is no corresponding (S,G) or (*,G)
entry. Furthermore, the mapping function is invoked by all routers
upon receiving a (*,G) or (*,*,RP) Join/Prune message.

Note that if the set of possible group-range-to-RP mappings changes,
each router will need to check whether any existing groups are
affected. This may, for example, cause a DR or acting DR to re-join
a group, or cause it to restart register encapsulation to the new RP.

   Implementation note: the bootstrap mechanism described in RFC 2362
   omitted step 1 above. However, of the implementations we are aware
   of, approximately half performed step 1 anyway. Note that
   implementations of BSR that omit step 1 will not correctly
   interoperate with implementations of this specification when used
   with the BSR mechanism described in [11].

4.7.2. Hash Function

The hash function is used by all routers within a domain, to map a
group to one of the RPs from the matching set of group-range-to-RP
mappings (this set all have the same longest mask length and same
highest priority). The algorithm takes as input the group address,
and the addresses of the candidate RPs from the mappings, and gives
as output one RP address to be used.

The protocol requires that all routers hash to the same RP within a
domain (except for transients). The following hash function must be
used in each router:

1. For RP addresses in the matching group-range-to-RP mappings,
    compute a value:

$$\text{Value}(G,M,C(i)) = (1103515245 * ((1103515245 * (G\&M) + 12345) \text{ XOR } C(i)) + 12345) \bmod 2^{31}$$

    where $C(i)$ is the RP address and M is a hash-mask. If BSR is
    being used, the hash-mask is given in the Bootstrap messages. If
    BSR is not being used, the alternative mechanism that supplies
    the group-range-to-RP mappings may supply the value, or else it
    defaults to a mask with the most significant 30 bits being one
    for IPv4 and the most significant 126 bits being one for IPv6.
    The hash-mask allows a small number of consecutive groups (e.g.,
    4) to always hash to the same RP. For instance, hierarchically-
    encoded data can be sent on consecutive group addresses to get
    the same delay and fate-sharing characteristics.

For address families other than IPv4, a 32-bit digest to be used
as C(i) and G must first be derived from the actual RP or group
address. Such a digest method must be used consistently
throughout the PIM domain. For IPv6 addresses, we recommend
using the equivalent IPv4 address for an IPv4-compatible address,
and the exclusive-or of each 32-bit segment of the address for
all other IPv6 addresses. For example, the digest of the IPv6
address 3ffe:b00:c18:1::10 would be computed as 0x3ffe0b00 ^
0x0c180001 ^ 0x00000000 ^ 0x00000010, where ^ represents the
exclusive-or operation.

2. The candidate RP with the highest resulting hash value is then
    the RP chosen by this Hash Function. If more than one RP has the
    same highest hash value, the RP with the highest IP address is
    chosen.

4.8. Source-Specific Multicast

The Source-Specific Multicast (SSM) service model [6] can be
implemented with a strict subset of the PIM-SM protocol mechanisms.
Both regular IP Multicast and SSM semantics can coexist on a single
router, and both can be implemented using the PIM-SM protocol. A
range of multicast addresses, currently 232.0.0.0/8 in IPv4 and
FF3x::/32 for IPv6, is reserved for SSM, and the choice of semantics
is determined by the multicast group address in both data packets and
PIM messages.

4.8.1. Protocol Modifications for SSM Destination Addresses

The following rules override the normal PIM-SM behavior for a
multicast address G in the SSM range:

o A router MUST NOT send a (*,G) Join/Prune message for any reason.

o A router MUST NOT send an (S,G,rpt) Join/Prune message for any
reason.

o A router MUST NOT send a Register message for any packet that is
  destined to an SSM address.

o A router MUST NOT forward packets based on (*,G) or (S,G,rpt)
  state. The (*,G)- and (S,G,rpt)-related state summarization macros
  are NULL for any SSM address, for the purposes of packet
  forwarding.

o A router acting as an RP MUST NOT forward any Register-encapsulated
  packet that has an SSM destination address.

The last two rules are present to deal with "legacy" routers unaware
of SSM that may be sending (*,G) and (S,G,rpt) Join/Prunes, or
Register messages for SSM destination addresses.

Additionally:

o A router MAY be configured to advertise itself as a Candidate RP
  for an SSM address. If so, it SHOULD respond with a Register-Stop
  message to any Register message containing a packet destined for an
  SSM address.

o A router MAY optimize out the creation and maintenance of (S,G,rpt)
  and (*,G) state for SSM destination addresses -- this state is not
  needed for SSM packets.

4.8.2. PIM-SSM-Only Routers

An implementer may choose to implement only the subset of PIM
Sparse-Mode that provides SSM forwarding semantics.

A PIM-SSM-only router MUST implement the following portions of this
specification:

o Upstream (S,G) state machine (Section 4.5.7)

o Downstream (S,G) state machine (Section 4.5.3)

o (S,G) Assert state machine (Section 4.6.1)

o Hello messages, neighbor discovery, and DR election (Section 4.3)

o Packet forwarding rules (Section 4.2)

A PIM-SSM-only router does not need to implement the following
protocol elements:

o Register state machine (Section 4.4)

o (*,G), (S,G,rpt), and (*,*,RP) Downstream state machines (Sections
  4.5.2, 4.5.4, and 4.5.1)

o (*,G), (S,G,rpt), and (*,*,RP) Upstream state machines (Sections
  4.5.6, 4.5.8, and 4.5.5)

o (*,G) Assert state machine (Section 4.6.2)

o Bootstrap RP Election (Section 4.7)

   o Keepalive Timer

   o SPTbit (Section 4.2.2)

   The Keepalive Timer should be treated as always running, and SPTbit
   should be treated as always being set for an SSM address.
   Additionally, the Packet forwarding rules of Section 4.2 can be
   simplified in a PIM-SSM-only router:

```
  if( iif == RPF_interface(S) AND UpstreamJPState(S,G) == Joined ) {
      oiflist = inherited_olist(S,G)
  } else if( iif is in inherited_olist(S,G) ) {
      send Assert(S,G) on iif
  }

  oiflist = oiflist (-) iif
  forward packet on all interfaces in oiflist
```

   This is nothing more than the reduction of the normal PIM-SM
   forwarding rule, with all (S,G,rpt) and (*,G) clauses replaced with
   NULL.

4.9. PIM Packet Formats

   This section describes the details of the packet formats for PIM
   control messages.

   All PIM control messages have IP protocol number 103.

   PIM messages are either unicast (e.g., Registers and Register-Stop)
   or multicast with TTL 1 to the 'ALL-PIM-ROUTERS' group (e.g.,
   Join/Prune, Asserts, etc.). The source address used for unicast
   messages is a domain-wide reachable address; the source address used
   for multicast messages is the link-local address of the interface on
   which the message is being sent.

   The IPv4 'ALL-PIM-ROUTERS' group is '224.0.0.13'. The IPv6 'ALL-PIM-
   ROUTERS' group is 'ff02::d'.

The PIM header common to all PIM messages is:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|PIM Ver| Type  |   Reserved    |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
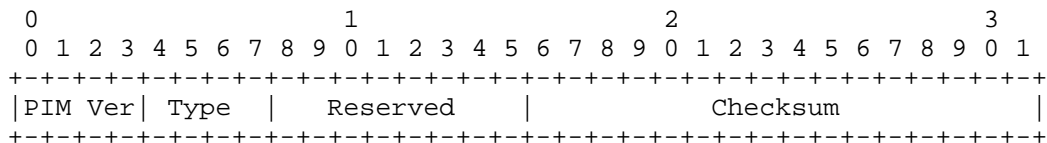
PIM Ver
     PIM Version number is 2.

Type Types for specific PIM messages. PIM Types are:

Message Type   Destination
----
0 = Hello   Multicast to ALL-PIM-ROUTERS
1 = Register   Unicast to RP
2 = Register-Stop   Unicast to source of Register
                                           packet
3 = Join/Prune   Multicast to ALL-PIM-ROUTERS
4 = Bootstrap   Multicast to ALL-PIM-ROUTERS
5 = Assert   Multicast to ALL-PIM-ROUTERS
6 = Graft (used in PIM-DM only)   Unicast to RPF'(S)
7 = Graft-Ack (used in PIM-DM only)   Unicast to source of Graft
                                           packet
8 = Candidate-RP-Advertisement   Unicast to Domain's BSR

Reserved
     Set to zero on transmission. Ignored upon receipt.

Checksum
     The checksum is a standard IP checksum, i.e., the 16-bit one's
     complement of the one's complement sum of the entire PIM
     message, excluding the "Multicast data packet" section of the
     Register message. For computing the checksum, the checksum
     field is zeroed. If the packet's length is not an integral
     number of 16-bit words, the packet is padded with a trailing
     byte of zero before performing the checksum.

     For IPv6, the checksum also includes the IPv6 "pseudo-header",
     as specified in RFC 2460, Section 8.1 [5]. This "pseudo-header"
     is prepended to the PIM header for the purposes of calculating
     the checksum. The "Upper-Layer Packet Length" in the pseudo-
     header is set to the length of the PIM message, except in
     Register messages where it is set to the length of the PIM
     register header (8). The Next Header value used in the pseudo-
     header is 103.

If a message is received with an unrecognized PIM Ver or Type field,
or if a message's destination does not correspond to the table above,
the message MUST be discarded, and an error message SHOULD be logged
to the administrator in a rate-limited manner.

4.9.1. Encoded Source and Group Address Formats

Encoded-Unicast Address

An Encoded-Unicast address takes the following format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Addr Family   | Encoding Type |      Unicast Address
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...
```
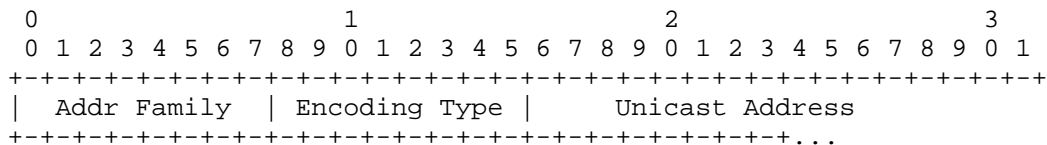
Addr Family
     The PIM address family of the 'Unicast Address' field of this
     address.

     Values 0-127 are as assigned by the IANA for Internet Address
     Families in [7]. Values 128-250 are reserved to be assigned by
     the IANA for PIM-specific Address Families. Values 251 though
     255 are designated for private use. As there is no assignment
     authority for this space, collisions should be expected.

Encoding Type
     The type of encoding used within a specific Address Family. The
     value '0' is reserved for this field and represents the native
     encoding of the Address Family.

Unicast Address
     The unicast address as represented by the given Address Family
     and Encoding Type.

Encoded-Group Address

Encoded-Group addresses take the following format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Addr Family  | Encoding Type |B| Reserved  |Z|   Mask Len    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Group multicast Address
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...
```
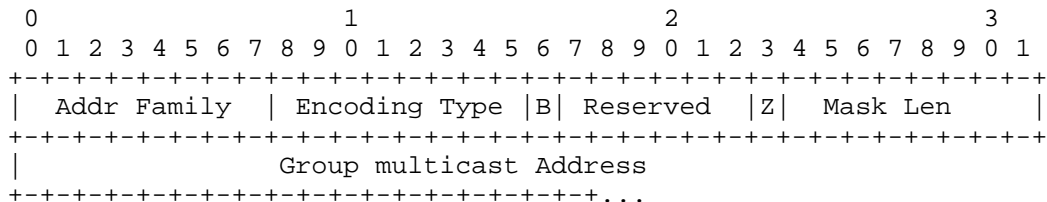
Addr Family
      Described above.

Encoding Type
      Described above.

[B]idirectional PIM
      Indicates the group range should use Bidirectional PIM [13].
      For PIM-SM defined in this specification, this bit MUST be zero.

 Reserved
      Transmitted as zero. Ignored upon receipt.

Admin Scope [Z]one
      indicates the group range is an admin scope zone. This is used
      in the Bootstrap Router Mechanism [11] only. For all other
      purposes, this bit is set to zero and ignored on receipt.

Mask Len
      The Mask length field is 8 bits. The value is the number of
      contiguous one bits that are left justified and used as a mask;
      when combined with the group address, it describes a range of
      groups. It is less than or equal to the address length in bits
      for the given Address Family and Encoding Type. If the message
      is sent for a single group, then the Mask length must equal the
      address length in bits for the given Address Family and Encoding
      Type (e.g., 32 for IPv4 native encoding, 128 for IPv6 native
      encoding).

Group multicast Address
      Contains the group address.

    Encoded-Source Address

    Encoded-Source address takes the following format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Addr Family   | Encoding Type | Rsrvd     |S|W|R|   Mask Len   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Source Address
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-...
```
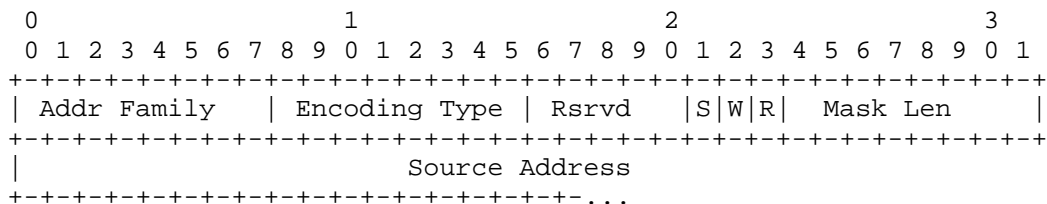
    Addr Family
         Described above.

    Encoding Type
         Described above.

    Reserved
         Transmitted as zero, ignored on receipt.

    S   The Sparse bit is a 1-bit value, set to 1 for PIM-SM. It is
         used for PIM version 1 compatibility.

    W   The WC (or WildCard) bit is a 1-bit value for use with PIM
         Join/Prune messages (see Section 4.9.5.1).

    R   The RPT (or Rendezvous Point Tree) bit is a 1-bit value for use
         with PIM Join/Prune messages (see Section 4.9.5.1). If the WC
         bit is 1, the RPT bit MUST be 1.

    Mask Len
         The mask length field is 8 bits. The value is the number of
         contiguous one bits left justified used as a mask which,
         combined with the Source Address, describes a source subnet.
         The mask length MUST be equal to the mask length in bits for the
         given Address Family and Encoding Type (32 for IPv4 native and
         128 for IPv6 native). A router SHOULD ignore any messages
         received with any other mask length.

    Source Address
         The source address.

4.9.2. Hello Message Format

   It is sent periodically by routers on all interfaces.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|PIM Ver| Type  |   Reserved    |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          OptionType           |         OptionLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         OptionValue                           |
|                            ...                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                              |
|                             .                                |
|                             .                                |
|                             .                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          OptionType           |         OptionLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         OptionValue                           |
|                            ...                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
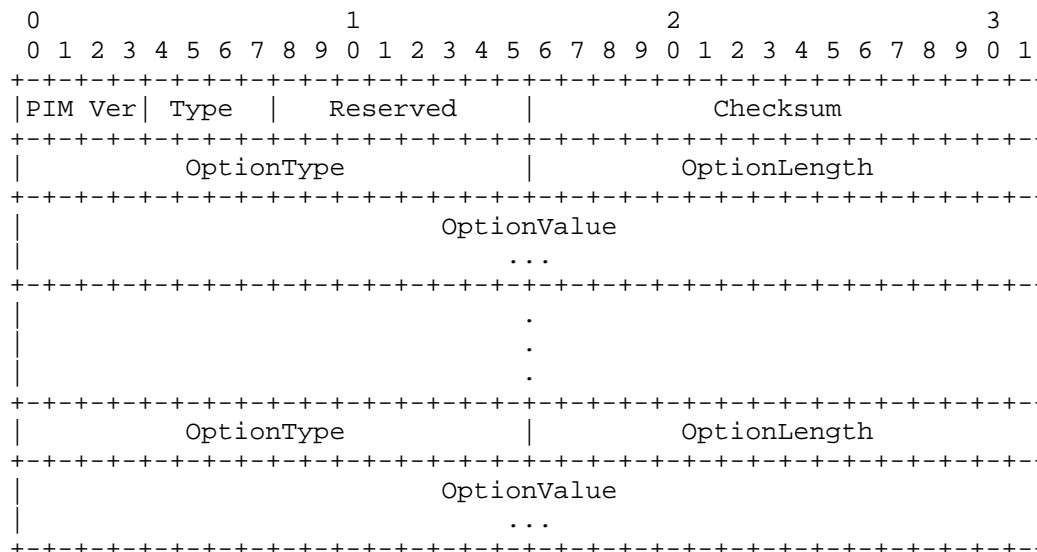
   PIM Version, Type, Reserved, Checksum
        Described in Section 4.9.

   OptionType
        The type of the option given in the following OptionValue field.

   OptionLength
        The length of the OptionValue field in bytes.

   OptionValue
        A variable length field, carrying the value of the option.

The Option fields may contain the following values:

o OptionType 1: Holdtime

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Type = 1           |            Length = 2         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Holdtime           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
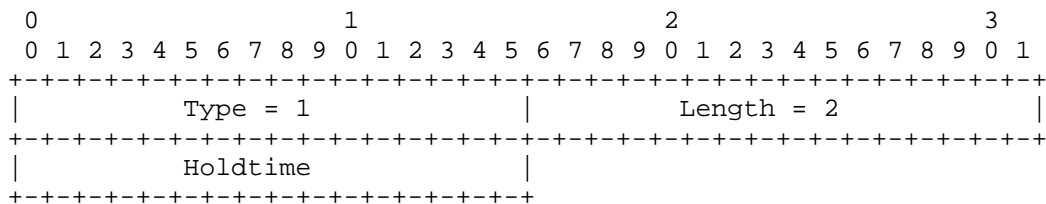
Holdtime is the amount of time a receiver must keep the neighbor
reachable, in seconds. If the Holdtime is set to '0xffff', the
receiver of this message never times out the neighbor. This may be
used with dial-on-demand links, to avoid keeping the link up with
periodic Hello messages.

Hello messages with a Holdtime value set to '0' are also sent by a
router on an interface about to go down or changing IP address (see
Section 4.3.1). These are effectively goodbye messages, and the
receiving routers should immediately time out the neighbor
information for the sender.

o OptionType 2: LAN Prune Delay

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Type = 2           |            Length = 4         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|T|     Propagation_Delay       |        Override_Interval      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
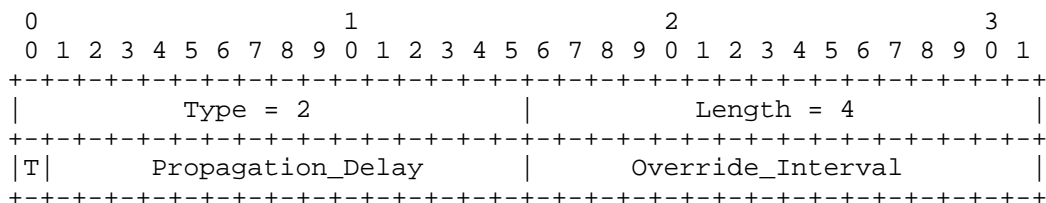
The LAN Prune Delay option is used to tune the prune propagation
delay on multi-access LANs. The T bit specifies the ability of the
sending router to disable joins suppression. Propagation_Delay and
Override_Interval are time intervals in units of milliseconds. A
router originating a LAN Prune Delay option on interface I sets the
Propagation_Delay field to the configured value of
Propagation_Delay(I) and the value of the Override_Interval field
to the value of Override_Interval(I). On a receiving router, the
values of the fields are used to tune the value of the
Effective_Override_Interval(I) and its derived timer values.
Section 4.3.3 describes how these values affect the behavior of a
router.

   o OptionType 3 to 16: reserved to be defined in future versions of
     this document.

   o OptionType 18: deprecated and should not be used.

   o OptionType 19: DR Priority

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |            Type = 19          |           Length = 4          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                          DR Priority                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
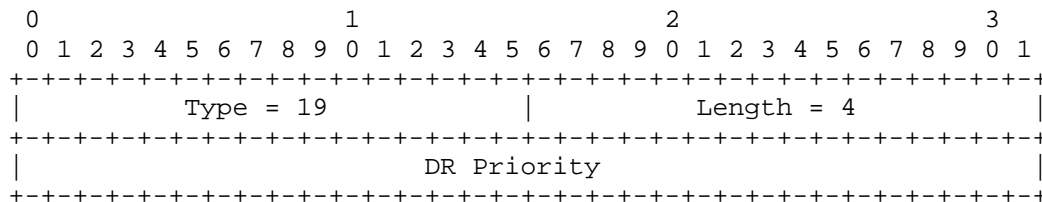
   DR Priority is a 32-bit unsigned number and should be considered in
   the DR election as described in Section 4.3.2.

   o OptionType 20: Generation ID

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |            Type = 20          |           Length = 4          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                         Generation ID                         |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
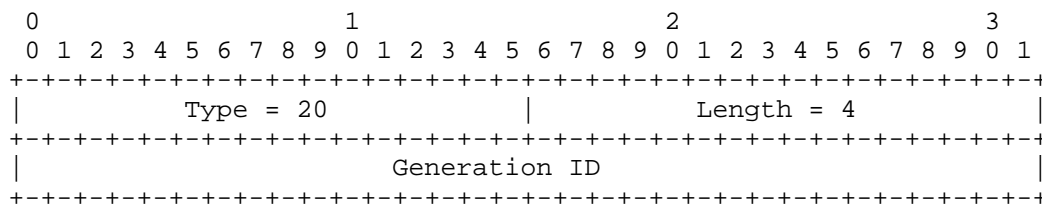
   Generation ID is a random 32-bit value for the interface on which
   the Hello message is sent. The Generation ID is regenerated
   whenever PIM forwarding is started or restarted on the interface.

   o OptionType 24: Address List

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |           Type = 24           |     Length = <Variable>       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |         Secondary Address 1 (Encoded-Unicast format)          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                  ...
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |         Secondary Address N (Encoded-Unicast format)          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
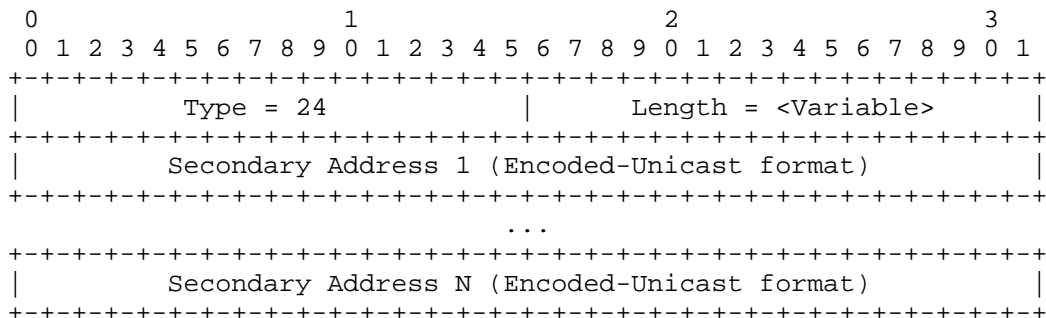
   The contents of the Address List Hello option are described in
   Section 4.3.4. All addresses within a single Address List must
   belong to the same address family.

   OptionTypes 17 through 65000 are assigned by the IANA. OptionTypes
   65001 through 65535 are reserved for Private Use, as defined in [9].

   Unknown options MUST be ignored and MUST NOT prevent a neighbor
   relationship from being formed. The "Holdtime" option MUST be
   implemented; the "DR Priority" and "Generation ID" options SHOULD be
   implemented. The "Address List" option MUST be implemented for IPv6.

4.9.3. Register Message Format

   A Register message is sent by the DR or a PMBR to the RP when a
   multicast packet needs to be transmitted on the RP-tree. The IP
   source address is set to the address of the DR, the destination
   address to the RP's address. The IP TTL of the PIM packet is the
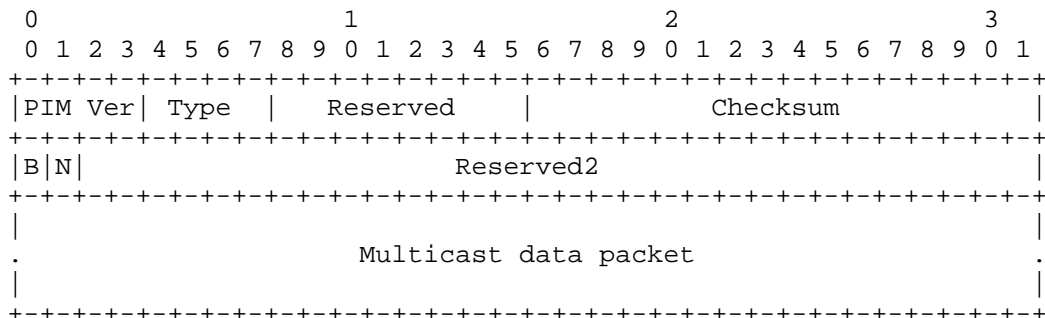   system's normal unicast TTL.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |PIM Ver| Type  |   Reserved    |           Checksum            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |B|N|                        Reserved2                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                               |
   .                     Multicast data packet                     .
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

PIM Version, Type, Reserved, Checksum
     Described in Section 4.9. Note that in order to reduce
     encapsulation overhead, the checksum for Registers is done only
     on the first 8 bytes of the packet, including the PIM header and
     the next 4 bytes, excluding the data packet portion. For
     interoperability reasons, a message carrying a checksum
     calculated over the entire PIM Register message should also be
     accepted. When calculating the checksum, the IPv6 pseudoheader
     "Upper-Layer Packet Length" is set to 8.

B   The Border bit. If the router is a DR for a source that it is
     directly connected to, it sets the B bit to 0. If the router is
     a PMBR for a source in a directly connected cloud, it sets the B
     bit to 1.

N   The Null-Register bit. Set to 1 by a DR that is probing the RP
     before expiring its local Register-Suppression Timer. Set to 0
     otherwise.

Reserved2
     Transmitted as zero, ignored on receipt.

Multicast data packet
     The original packet sent by the source. This packet must be of
     the same address family as the encapsulating PIM packet, e.g.,
     an IPv6 data packet must be encapsulated in an IPv6 PIM packet.
     Note that the TTL of the original packet is decremented before
     encapsulation, just like any other packet that is forwarded. In
     addition, the RP decrements the TTL after decapsulating, before
     forwarding the packet down the shared tree.

     For (S,G) Null-Registers, the Multicast data packet portion
     contains a dummy IP header with S as the source address, G as
     the destination address. When generating an IPv4 Null-Register
     message, the fields in the dummy IPv4 header SHOULD be filled in
     according to the following table. Other IPv4 header fields may
     contain any value that is valid for that field.

Field	Value
IP Version	4
Header Length	5
Checksum	Header checksum
Fragmentation offset	0
More Fragments	0
Total Length	20
IP Protocol	103 (PIM)

On receipt of an (S,G) Null-Register, if the Header Checksum
field is non-zero, the recipient SHOULD check the checksum and
discard null registers that have a bad checksum. The recipient
SHOULD NOT check the value of any individual fields; a correct
IP header checksum is sufficient. If the Header Checksum field
is zero, the recipient MUST NOT check the checksum.

With IPv6, an implementation generates a dummy IP header
followed by a dummy PIM header with values according to the
following table in addition to the source and group. Other IPv6
header fields may contain any value that is valid for that
field.

```
Header Field   Value
-------------------------------------
IP Version     6
Next Header    103 (PIM)
Length         4
PIM Version    0
PIM Type       0
PIM Reserved   0
PIM Checksum   PIM checksum including
               IPv6 "pseudo-header";
               see Section 4.9
```

On receipt of an IPv6 (S,G) Null-Register, if the dummy PIM
header is present, the recipient SHOULD check the checksum and
discard Null-Registers that have a bad checksum.

4.9.4. Register-Stop Message Format

   A Register-Stop is unicast from the RP to the sender of the Register
   message. The IP source address is the address to which the register
   was addressed. The IP destination address is the source address of
   the register message.
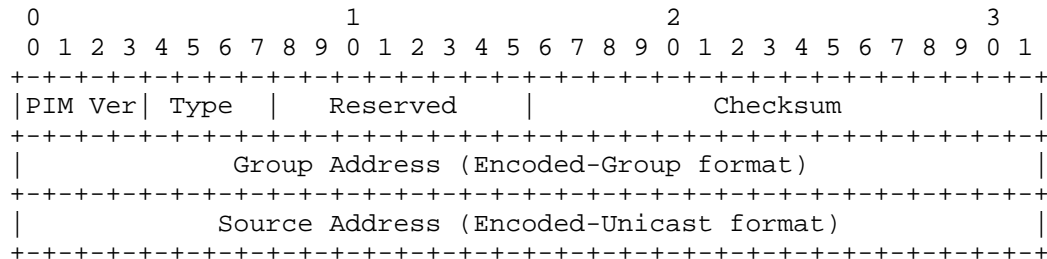
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|PIM Ver| Type  |   Reserved    |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Group Address (Encoded-Group format)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Source Address (Encoded-Unicast format)           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

   PIM Version, Type, Reserved, Checksum
        Described in Section 4.9.
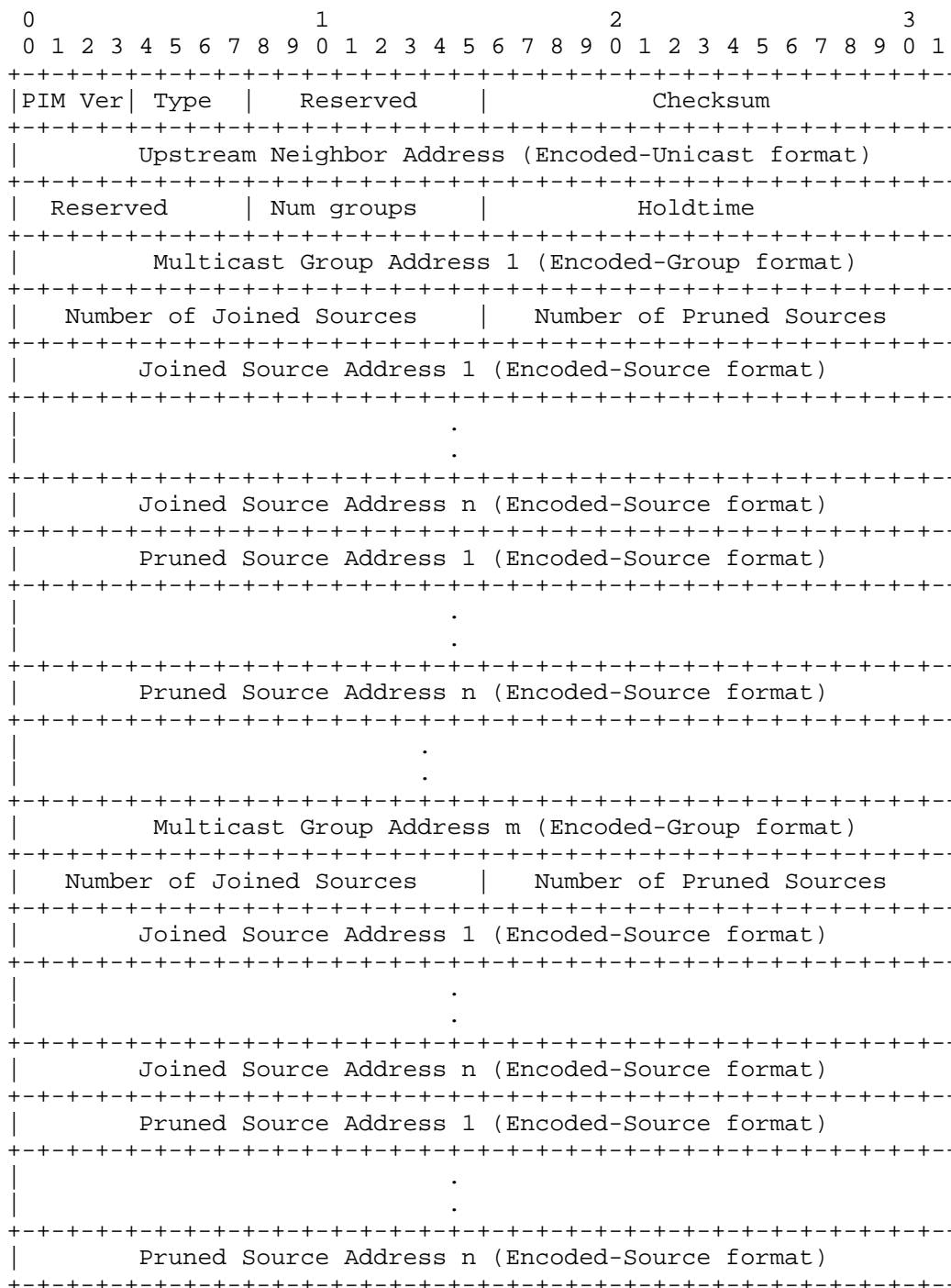
   Group Address
        The group address from the multicast data packet in the
        Register. Format described in Section 4.9.1. Note that for
        Register-Stops the Mask Len field contains the full address
        length * 8 (e.g., 32 for IPv4 native encoding), if the message
        is sent for a single group.

   Source Address
        The host address of the source from the multicast data packet in
        the register. The format for this address is given in the
        Encoded-Unicast address in Section 4.9.1. A special wild card
        value consisting of an address field of all zeros can be used to
        indicate any source.

4.9.5. Join/Prune Message Format

   A Join/Prune message is sent by routers towards upstream sources and
   RPs. Joins are sent to build shared trees (RP trees) or source trees
   (SPT). Prunes are sent to prune source trees when members leave
   groups as well as sources that do not use the shared tree.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|PIM Ver| Type  |   Reserved    |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Upstream Neighbor Address (Encoded-Unicast format)      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Reserved     | Num groups    |          Holdtime             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Multicast Group Address 1 (Encoded-Group format)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Number of Joined Sources    |   Number of Pruned Sources    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Joined Source Address 1 (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               .                               |
|                               .                               |
|                               .                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Joined Source Address n (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Pruned Source Address 1 (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               .                               |
|                               .                               |
|                               .                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Pruned Source Address n (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               .                               |
|                               .                               |
|                               .                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Multicast Group Address m (Encoded-Group format)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Number of Joined Sources    |   Number of Pruned Sources    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Joined Source Address 1 (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               .                               |
|                               .                               |
|                               .                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Joined Source Address n (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Pruned Source Address 1 (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               .                               |
|                               .                               |
|                               .                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Pruned Source Address n (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

PIM Version, Type, Reserved, Checksum
     Described in Section 4.9.

Unicast Upstream Neighbor Address
     The address of the upstream neighbor that is the target of the
     message. The format for this address is given in the Encoded-
     Unicast address in Section 4.9.1. For IPv6 the source address
     used for multicast messages is the link-local address of the
     interface on which the message is being sent. For IPv4, the
     source address is the primary address associated with that
     interface.

Reserved
     Transmitted as zero, ignored on receipt.

Holdtime
     The amount of time a receiver must keep the Join/Prune state
     alive, in seconds. If the Holdtime is set to '0xffff', the
     receiver of this message should hold the state until canceled by
     the appropriate canceling Join/Prune message, or timed out
     according to local policy. This may be used with dial-on-demand
     links, to avoid keeping the link up with periodic Join/Prune
     messages.

     Note that the HoldTime must be larger than the
     J/P_Override_Interval(I).

Number of Groups
     The number of multicast group sets contained in the message.

Multicast group address
     For format description, see Section 4.9.1.

Number of Joined Sources
     Number of joined source addresses listed for a given group.

Joined Source Address 1 .. n
     This list contains the sources for a given group that the
     sending router will forward multicast datagrams from if received
     on the interface on which the Join/Prune message is sent.

     See Encoded-Source-Address format in Section 4.9.1.

Number of Pruned Sources
     Number of pruned source addresses listed for a group.

     Pruned Source Address 1 .. n
          This list contains the sources for a given group that the
          sending router does not want to forward multicast datagrams from
          when received on the interface on which the Join/Prune message
          is sent.

     Within one PIM Join/Prune message, all the Multicast Group Addresses,
     Joined Source addresses, and Pruned Source addresses MUST be of the
     same address family. It is NOT PERMITTED to mix IPv4 and IPv6
     addresses within the same message. In addition, the address family
     of the fields in the message SHOULD be the same as the IP source and
     destination addresses of the packet. This permits maximum
     implementation flexibility for dual-stack IPv4/IPv6 routers. If a
     router receives a message with mixed family addresses, it SHOULD only
     process the addresses that are of the same family as the unicast
     upstream neighbor address.

4.9.5.1. Group Set Source List Rules

     As described above, Join/Prune messages are composed of one or more
     group sets. Each set contains two source lists, the Joined Sources
     and the Pruned Sources. This section describes the different types
     of group sets and source list entries that can exist in a Join/Prune
     message.

     There are two valid group set types:

     Wildcard Group Set
          The wildcard group set is represented by the entire multicast
          range: the beginning of the multicast address range in the
          group address field and the prefix length of the multicast
          address range in the mask length field of the Multicast Group
          Address (i.e., '224.0.0.0/4' for IPv4 or 'ff00::/8' for IPv6).
          Each Join/Prune message SHOULD contain at most one wildcard
          group set. Each wildcard group set may contain one or more
          (*,*,RP) source list entries in either the Joined or Pruned
          lists.

          A (*,*,RP) source list entry may only exist in a wildcard group
          set. When added to a Joined source list, this type of source
          entry expresses the router's interest in receiving traffic for
          all groups mapping to the specified RP. When added to a Pruned
          source list a (*,*,RP) entry expresses the router's interest to
          stop receiving such traffic. Note that as indicated by the
          Join/Prune state machines, such a Join or Prune will NOT
          override Join/Prune state created using a Group-Specific Set
          (see below).

        (*,*,RP) source list entries have the Source-Address set to the
        address of the RP, the Source-Address Mask-Len set to the full
        length of the IP address, and both the WC and RPT bits of the
        Source-Address set to 1.

   Group-Specific Set
        A Group-Specific Set is represented by a valid IP multicast
        address in the group address field and the full length of the IP
        address in the mask length field of the Multicast Group Address.
        Each Join/Prune message SHOULD NOT contain more than one group-
        specific set for the same IP multicast address. Each group-
        specific set may contain (*,G), (S,G,rpt), and (S,G) source list
        entries in the Joined or Pruned lists.

     (*,G)
        The (*,G) source list entry is used in Join/Prune messages
        sent towards the RP for the specified group. It expresses
        interest (or lack thereof) in receiving traffic sent to the
        group through the Rendezvous-Point shared tree. There may
        only be one such entry in both the Joined and Pruned lists of
        a group-specific set.

        (*,G) source list entries have the Source-Address set to the
        address of the RP for group G, the Source-Address Mask-Len set
        to the full length of the IP address, and both the WC and RPT
        bits of the Encoded-Source-Address set.

     (S,G,rpt)
        The (S,G,rpt) source list entry is used in Join/Prune messages
        sent towards the RP for the specified group. It expresses
        interest (or lack thereof) in receiving traffic through the
        shared tree sent by the specified source to this group. For
        each source address, the entry may exist in only one of the
        Joined and Pruned source lists of a group-specific set, but
        not both.

        (S,G,rpt) source list entries have the Source-Address set to
        the address of the source S, the Source-Address Mask-Len set
        to the full length of the IP address, and the WC bit cleared
        and the RPT bit set in the Encoded-Source-Address.

     (S,G)
        The (S,G) source list entry is used in Join/Prune messages
        sent towards the specified source. It expresses interest (or
        lack thereof) in receiving traffic through the shortest path
        tree sent by the source to the specified group. For each
        source address, the entry may exist in only one of the Joined
        and Pruned source lists of a group-specific set, but not both.

          (S,G) source list entries have the Source-Address set to the
          address of the source S, the Source-Address Mask-Len set to
          the full length of the IP address, and both the WC and RPT
          bits of the Encoded-Source-Address cleared.

   The rules described above are sufficient to prevent invalid
   combinations of source list entries in group-specific sets. There
   are, however, a number of combinations that have a valid
   interpretation but that are not generated by the protocol as
   described in this specification:

   o Combining a (*,G) Join and a (S,G,rpt) Join entry in the same
     message is redundant as the (*,G) entry covers the information
     provided by the (S,G,rpt) entry.

   o The same applies for a (*,G) Prunes and (S,G,rpt) Prunes.

   o The combination of a (*,G) Prune and a (S,G,rpt) Join is also not
     generated. (S,G,rpt) Joins are only sent when the router is
     receiving all traffic for a group on the shared tree and it wishes
     to indicate a change for the particular source. As a (*,G) prune
     indicates that the router no longer wishes to receive shared tree
     traffic, the (S,G,rpt) Join would be meaningless.

   o As Join/Prune messages are targeted to a single PIM neighbor,
     including both a (S,G) Join and a (S,G,rpt) Prune in the same
     message is usually redundant. The (S,G) Join informs the neighbor
     that the sender wishes to receive the particular source on the
     shortest path tree. It is therefore unnecessary for the router to
     say that it no longer wishes to receive it on the shared tree.
     However, there is a valid interpretation for this combination of
     entries. A downstream router may have to instruct its upstream
     only to start forwarding a specific source once it has started
     receiving the source on the shortest-path tree.

   o The combination of a (S,G) Prune and a (S,G,rpt) Join could
     possibly be used by a router to switch from receiving a particular
     source on the shortest-path tree back to receiving it on the shared
     tree (provided that the RPF neighbor for the shortest-path and
     shared trees is common). However, Sparse-Mode PIM does not provide
     a mechanism for explicitly switching back to the shared tree.

The rules are summarized in the tables below.

		Join (*,G)	Prune (*,G)	Join (S,G,rpt)	Prune (S,G,rpt)	Join (S,G)	Prune (S,G)
Join (*,G)		-	no	?	yes	yes	yes
Prune (*,G)		no	-	?	?	yes	yes
Join (S,G,rpt)		?	?	-	no	yes	?
Prune (S,G,rpt)		yes	?	no	-	yes	?
Join (S,G)		yes	yes	yes	yes	-	no
Prune (S,G)		yes	yes	?	?	no	-

		Join (*,*,RP)	Prune (*,*,RP)	all others
Join (*,*,RP)		-	no	yes
Prune (*,*,RP)		no	-	yes
all others		yes	yes	see above

yes Allowed and expected.

no   Combination is not allowed by the protocol and MUST NOT be
     generated by a router. A router MAY accept these messages, but
     the result is undefined. An error message MAY be logged to the
     administrator in a rate-limited manner.

?   Combination not expected by the protocol, but well-defined. A
     router MAY accept it but SHOULD NOT generate it.

The order of source list entries in a group set source list is not
important, except where limited by the packet format itself.

4.9.5.2. Group Set Fragmentation

   When building a Join/Prune for a particular neighbor, a router should
   try to include in the message as much of the information it needs to
   convey to the neighbor as possible. This implies adding one group
   set for each multicast group that has information pending
   transmission and within each set including all relevant source list
   entries.

   On a router with a large amount of multicast state, the number of
   entries that must be included may result in packets that are larger
   than the maximum IP packet size. In most such cases, the information
   may be split into multiple messages.

   There is an exception with group sets that contain a (*,G) Joined
   source list entry. The group set expresses the router's interest in
   receiving all traffic for the specified group on the shared tree, and
   it MUST include an (S,G,rpt) Pruned source list entry for every
   source that the router does not wish to receive. This list of
   (S,G,rpt) Pruned source-list entries MUST not be split in multiple
   messages.

   If only N (S,G,rpt) Prune entries fit into a maximum-sized Join/Prune
   message, but the router has more than N (S,G,rpt) Prunes to add, then
   the router MUST choose to include the first N (numerically smallest
   in network byte order) IP addresses.

4.9.6. Assert Message Format

   The Assert message is used to resolve forwarder conflicts between
   routers on a link. It is sent when a router receives a multicast
   data packet on an interface on which the router would normally have
   forwarded that packet. Assert messages may also be sent in response
   to an Assert message from another router.
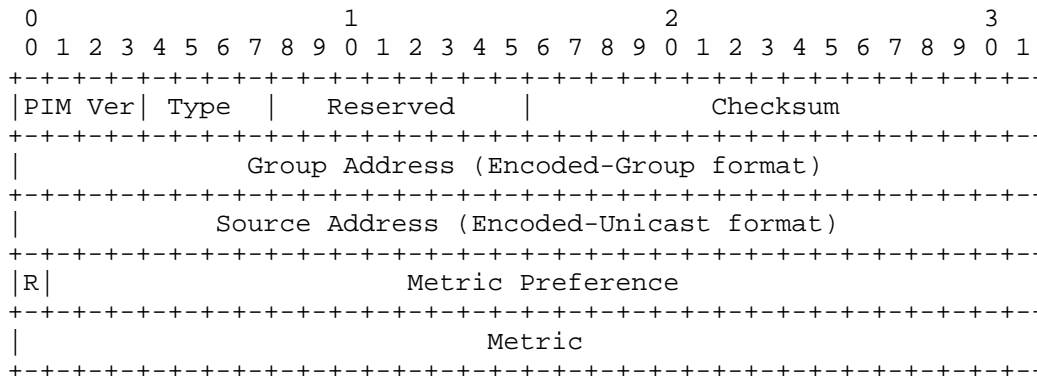
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|PIM Ver| Type  |   Reserved    |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Group Address (Encoded-Group format)              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Source Address (Encoded-Unicast format)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R|                   Metric Preference                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Metric                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

PIM Version, Type, Reserved, Checksum
     Described in Section 4.9.

Group Address
     The group address for which the router wishes to resolve the
     forwarding conflict. This is an Encoded-Group address, as
     specified in Section 4.9.1.

Source Address
     Source address for which the router wishes to resolve the
     forwarding conflict. The source address MAY be set to zero for
     (*,G) asserts (see below). The format for this address is given
     in Encoded-Unicast-Address in Section 4.9.1.

R   RPT-bit is a 1-bit value. The RPT-bit is set to 1 for
     Assert(*,G) messages and 0 for Assert(S,G) messages.

Metric Preference
     Preference value assigned to the unicast routing protocol that
     provided the route to the multicast source or Rendezvous-Point.

Metric
     The unicast routing table metric associated with the route used
     to reach the multicast source or Rendezvous-Point. The metric
     is in units applicable to the unicast routing protocol used.

Assert messages can be sent to resolve a forwarding conflict for all
traffic to a given group or for a specific source and group.

     Assert(S,G)
          Source-specific asserts are sent by routers forwarding a
          specific source on the shortest-path tree (SPTbit is TRUE).
          (S,G) Asserts have the Group-Address field set to the group G
          and the Source-Address field set to the source S. The RPT-bit
          is set to 0, the Metric-Preference is set to MRIB.pref(S) and
          the Metric is set to MRIB.metric(S).

     Assert(*,G)
          Group-specific asserts are sent by routers forwarding data for
          the group and source(s) under contention on the shared tree.
          (*,G) asserts have the Group-Address field set to the group G.
          For data-triggered Asserts, the Source-Address field MAY be set
          to the IP source address of the data packet that triggered the
          Assert and is set to zero otherwise. The RPT-bit is set to 1,
          the Metric-Preference is set to MRIB.pref(RP(G)), and the Metric
          is set to MRIB.metric(RP(G)).

4.10. PIM Timers

   PIM-SM maintains the following timers, as discussed in Section 4.1.
   All timers are countdown timers; they are set to a value and count
   down to zero, at which point they typically trigger an action. Of
   course they can just as easily be implemented as count-up timers,
   where the absolute expiry time is stored and compared against a
   real-time clock, but the language in this specification assumes that
   they count downwards to zero.

   Global Timers

   Per interface (I):

          Hello Timer: HT(I)

          Per neighbor (N):

               Neighbor Liveness Timer: NLT(N,I)

          Per active RP (RP):

               (*,*,RP) Join Expiry Timer: ET(*,*,RP,I)

               (*,*,RP) Prune-Pending Timer: PPT(*,*,RP,I)

          Per Group (G):

               (*,G) Join Expiry Timer: ET(*,G,I)

               (*,G) Prune-Pending Timer: PPT(*,G,I)

               (*,G) Assert Timer: AT(*,G,I)

          Per Source (S):

               (S,G) Join Expiry Timer: ET(S,G,I)

               (S,G) Prune-Pending Timer: PPT(S,G,I)

               (S,G) Assert Timer: AT(S,G,I)

               (S,G,rpt) Prune Expiry Timer: ET(S,G,rpt,I)

               (S,G,rpt) Prune-Pending Timer: PPT(S,G,rpt,I)

   Per active RP (RP):

       (*,*,RP) Upstream Join Timer: JT(*,*,RP)

   Per Group (G):

       (*,G) Upstream Join Timer: JT(*,G)

        Per Source (S):

            (S,G) Upstream Join Timer: JT(S,G)

            (S,G) Keepalive Timer: KAT(S,G)

            (S,G,rpt) Upstream Override Timer: OT(S,G,rpt)

   At the DRs or relevant Assert Winners only:

       Per Source,Group pair (S,G):

            Register-Stop Timer: RST(S,G)

4.11. Timer Values

   When timers are started or restarted, they are set to default values.
   This section summarizes those default values.

   Note that protocol events or configuration may change the default
   value of a timer on a specific interface. When timers are
   initialized in this document, the value specific to the interface in
   context must be used.

Some of the timers listed below (Prune-Pending, Upstream Join,
Upstream Override) can be set to values that depend on the settings
of the Propagation_Delay and Override_Interval of the corresponding
interface. The default values for these are given below.

Variable Name: Propagation_Delay(I)

+------------------------------+-------------+----------------------+
| Value Name | Value | Explanation |
+------------------------------+-------------+----------------------+
Propagation_delay_default	0.5 secs	Expected
		propagation delay
		over the local
		link.
+------------------------------+-------------+----------------------+

The default value of the Propagation_delay_default is chosen to be
relatively large to provide compatibility with older PIM
implementations.

Variable Name: Override_Interval(I)

+-------------------------+----------------+------------------------+
| Value Name | Value | Explanation |
+-------------------------+----------------+------------------------+
t_override_default	2.5 secs	Default delay
		interval over
		which to randomize
		when scheduling a
		delayed Join
		message.
+-------------------------+----------------+------------------------+

Timer Name: Hello Timer (HT(I))

+--------------------+--------+--------------------------------------+
|Value Name | Value | Explanation |
+--------------------+--------+--------------------------------------+
|Hello_Period | 30 secs| Periodic interval for Hello messages.|
+--------------------+--------+--------------------------------------+
Triggered_Hello_Delay	5 secs	Randomized interval for initial Hello
		message on bootup or triggered Hello
		message to a rebooting neighbor.
+--------------------+--------+--------------------------------------+

At system power-up, the timer is initialized to rand(0,
Triggered_Hello_Delay) to prevent synchronization. When a new or
rebooting neighbor is detected, a responding Hello is sent within
rand(0, Triggered_Hello_Delay).

Timer Name: Neighbor Liveness Timer (NLT(N,I))

Value Name	Value	Explanation
Default_Hello_Holdtime	3.5 * Hello_Period	Default holdtime to keep neighbor state alive
Hello_Holdtime	from message	Holdtime from Hello Message Holdtime option.

The Holdtime in a Hello Message should be set to (3.5 *
Hello_Period), giving a default value of 105 seconds.

Timer Names: Expiry Timer (ET(*,*,RP,I), ET(*,G,I), ET(S,G,I),
ET(S,G,rpt,I))

Value Name	Value	Explanation
J/P_HoldTime	from message	Holdtime from Join/Prune Message

See details of JT(*,G) for the Holdtime that is included in
Join/Prune Messages.

Timer Names: Prune-Pending Timer (PPT(*,*,RP,I), PPT(*,G,I),
PPT(S,G,I), PPT(S,G,rpt,I))

+--------------------------+--------------------+--------------------+
|Value Name | Value | Explanation |
+--------------------------+--------------------+--------------------+
J/P_Override_Interval(I)	Default:	Short period after
	Effective_	a join or prune to
	Propagation_	allow other
	Delay(I) +	routers on the LAN
	EffectiveOverride_	to override the
	Interval(I)	join or prune
+--------------------------+--------------------+--------------------+

Note that both the Effective_Propagation_Delay(I) and the
Effective_Override_Interval(I) are interface-specific values that may
change when Hello messages are received (see Section 4.3.3).

Timer Names: Assert Timer (AT(*,G,I), AT(S,G,I))

+--------------------------+--------------------+-------------------+
| Value Name | Value | Explanation |
+--------------------------+--------------------+-------------------+
Assert_Override_Interval	Default: 3 secs	Short interval
		before an assert
		times out where
		the assert winner
		resends an Assert
		message
+--------------------------+--------------------+-------------------+
Assert_Time	Default: 180 secs	Period after last
		assert before
		assert state is
		timed out
+--------------------------+--------------------+-------------------+

Note that for historical reasons, the Assert message lacks a Holdtime
field. Thus, changing the Assert Time from the default value is not
recommended.

   Timer Names: Upstream Join Timer (JT(*,*,RP), JT(*,G), JT(S,G))

+-------------+-------------------+----------------------------------+
|Value Name | Value | Explanation |
+-------------+-------------------+----------------------------------+
|t_periodic | Default: 60 secs | Period between Join/Prune Messages|
+-------------+-------------------+----------------------------------+
t_suppressed	rand(1.1 *	Suppression period when someone
	t_periodic, 1.4 *	else sends a J/P message so we
	t_periodic) when	don't need to do so.
	Suppression_	
	Enabled(I) is	
	true, 0 otherwise	
+-------------+-------------------+----------------------------------+
t_override	rand(0, Effective_	Randomized delay to prevent
	Override_	response implosion when sending a
	Interval(I))	join message to override someone
		else's Prune message.
+-------------+-------------------+----------------------------------+

   t_periodic may be set to take into account such things as the
   configured bandwidth and expected average number of multicast route
   entries for the attached network or link (e.g., the period would be
   longer for lower-speed links, or for routers in the center of the
   network that expect to have a larger number of entries). If the
   Join/Prune-Period is modified during operation, these changes should
   be made relatively infrequently, and the router should continue to
   refresh at its previous Join/Prune-Period for at least Join/Prune-
   Holdtime, in order to allow the upstream router to adapt.

   The holdtime specified in a Join/Prune message should be set to (3.5
   * t_periodic).

   t_override depends on the Effective_Override_Interval of the upstream
   interface, which may change when Hello messages are received.

   t_suppressed depends on the Suppression State of the upstream
   interface (Section 4.3.3) and becomes zero when suppression is
   disabled.

   Timer Name: Upstream Override Timer (OT(S,G,rpt))

   +---------------+------------------------+--------------------------+
   | Value Name | Value | Explanation |
   +---------------+------------------------+--------------------------+
   | t_override | see Upstream Join Timer | see Upstream Join Timer |
   +---------------+------------------------+--------------------------+

   The upstream Override Timer is only ever set to t_override; this
   value is defined in the section on Upstream Join Timers.

   Timer Name: Keepalive Timer (KAT(S,G))

   +----------------------+---------------------+---------------------+
   | Value Name | Value | Explanation |
   +----------------------+---------------------+---------------------+
Keepalive_Period	Default: 210 secs	Period after last
		(S,G) data packet
		during which (S,G)
		Join state will be
		maintained even in
		the absence of
		(S,G) Join
		messages.
   +----------------------+---------------------+---------------------+
RP_Keepalive_Period	(3 * Register_	As
	Suppression_Time)	Keepalive_Period,
	+ Register_	but at the RP when
	Probe_Time	a Register-Stop is
		sent.
   +----------------------+---------------------+---------------------+

   The normal keepalive period for the KAT(S,G) defaults to 210 seconds.
   However, at the RP, the keepalive period must be at least the
   Register_Suppression_Time, or the RP may time out the (S,G) state
   before the next Null-Register arrives. Thus, the KAT(S,G) is set to
   max(Keepalive_Period, RP_Keepalive_Period) when a Register-Stop is
   sent.

    Timer Name: Register-Stop Timer (RST(S,G))

+---------------------------+-------------------+--------------------+
|Value Name | Value | Explanation |
+---------------------------+-------------------+--------------------+
Register_Suppression_Time	Default: 60 secs	Period during
		which a DR stops
		sending Register-
		encapsulated data
		to the RP after
		receiving a
		Register-Stop
		message.
+---------------------------+-------------------+--------------------+
Register_Probe_Time	Default: 5 secs	Time before RST
		expires when a DR
		may send a Null-
		Register to the RP
		to cause it to
		resend a Register-
		Stop message.
+---------------------------+-------------------+--------------------+

   If the Register_Suppression_Time or the Register_Probe_Time are
   configured to values other than the defaults, it MUST be ensured that
   the value of the Register_Probe_Time is less than half the value of
   the Register_Suppression_Time to prevent a possible negative value in
   the setting of the Register-Stop Timer.

5. IANA Considerations

5.1. PIM Address Family

   The PIM Address Family field was chosen to be 8 bits as a tradeoff
   between packet format and use of the IANA assigned numbers. Because
   when the PIM packet format was designed only 15 values were assigned
   for Address Families, and large numbers of new Address Family values
   were not envisioned, 8 bits seemed large enough. However, the IANA
   assigns Address Families in a 16-bit field. Therefore, the PIM
   Address Family is allocated as follows:

      Values 0 through 127 are designated to have the same meaning as
      IANA-assigned Address Family Numbers [7].

      Values 128 through 250 are designated to be assigned for PIM by the
      IANA based upon IESG Approval, as defined in [9].

      Values 251 through 255 are designated for Private Use, as defined

   in [9].

5.2. PIM Hello Options

   Values 17 through 65000 are to be assigned by the IANA. Since the
   space is large, they may be assigned as First Come First Served as
   defined in [9]. Such assignments are valid for one year and may be
   renewed. Permanent assignments require a specification (see
   "Specification Required" in [9].)

6. Security Considerations

   This section describes various possible security concerns related to
   the PIM-SM protocol, including a description of how to use IPsec to
   secure the protocol. The reader is referred to [15] and [16] for
   further discussion of PIM-SM and multicast security. The IPsec
   authentication header [8] MAY be used to provide data integrity
   protection and groupwise data origin authentication of PIM protocol
   messages. Authentication of PIM messages can protect against
   unwanted behaviors caused by unauthorized or altered PIM messages.

6.1. Attacks Based on Forged Messages

   The extent of possible damage depends on the type of counterfeit
   messages accepted. We next consider the impact of possible
   forgeries, including forged link-local (Join/Prune, Hello, and
   Assert) and forged unicast (Register and Register-Stop) messages.

6.1.1. Forged Link-Local Messages

   Join/Prune, Hello, and Assert messages are all sent to the link-local
   ALL_PIM_ROUTERS multicast addresses and thus are not forwarded by a
   compliant router. A forged message of this type can only reach a LAN
   if it was sent by a local host or if it was allowed onto the LAN by a
   compromised or non-compliant router.

   1. A forged Join/Prune message can cause multicast traffic to be
       delivered to links where there are no legitimate requesters,
       potentially wasting bandwidth on that link. A forged leave
       message on a multi-access LAN is generally not a significant
       attack in PIM, because any legitimately joined router on the LAN
       would override the leave with a join before the upstream router
       stops forwarding data to the LAN.

   2. By forging a Hello message, an unauthorized router can cause
       itself to be elected as the designated router on a LAN. The
       designated router on a LAN is (in the absence of asserts)
       responsible for forwarding traffic to that LAN on behalf of any

local members. The designated router is also responsible for
register-encapsulating to the RP any packets that are originated
by hosts on the LAN. Thus, the ability of local hosts to send
and receive multicast traffic may be compromised by a forged
Hello message.

3. By forging an Assert message on a multi-access LAN, an attacker
    could cause the legitimate designated forwarder to stop
    forwarding traffic to the LAN. Such a forgery would prevent any
    hosts downstream of that LAN from receiving traffic.

6.1.2. Forged Unicast Messages

Register messages and Register-Stop messages are forwarded by
intermediate routers to their destination using normal IP forwarding.
Without data origin authentication, an attacker who is located
anywhere in the network may be able to forge a Register or Register-
Stop message. We consider the effect of a forgery of each of these
messages next.

1. By forging a Register message, an attacker can cause the RP to
    inject forged traffic onto the shared multicast tree.

2. By forging a Register-stop message, an attacker can prevent a
    legitimate DR from Registering packets to the RP. This can
    prevent local hosts on that LAN from sending multicast packets.

The above two PIM messages are not changed by intermediate routers
and need only be examined by the intended receiver. Thus, these
messages can be authenticated end-to-end, using AH. Attacks on
Register and Register-Stop messages do not apply to a PIM-SSM-only
implementation, as these messages are not required for PIM-SSM.

6.2. Non-Cryptographic Authentication Mechanisms

A PIM router SHOULD provide an option to limit the set of neighbors
from which it will accept Join/Prune, Assert, and Hello messages.
Either static configuration of IP addresses or an IPsec security
association may be used. Furthermore, a PIM router SHOULD NOT accept
protocol messages from a router from which it has not yet received a
valid Hello message.

A Designated Router MUST NOT register-encapsulate a packet and send
it to the RP unless the source address of the packet is a legal
address for the subnet on which the packet was received. Similarly,
a Designated Router SHOULD NOT accept a Register-Stop packet whose IP
source address is not a valid RP address for the local domain.

An implementation SHOULD provide a mechanism to allow an RP to
restrict the range of source addresses from which it accepts
Register-encapsulated packets.

All options that restrict the range of addresses from which packets
are accepted MUST default to allowing all packets.

6.3. Authentication Using IPsec

The IPsec [8] transport mode using the Authentication Header (AH) is
the recommended method to prevent the above attacks against PIM. The
specific AH authentication algorithm and parameters, including the
choice of authentication algorithm and the choice of key, are
configured by the network administrator. When IPsec authentication
is used, a PIM router should reject (drop without processing) any
unauthorized PIM protocol messages.

To use IPsec, the administrator of a PIM network configures each PIM
router with one or more security associations (SAs) and associated
Security Parameter Indexes (SPIs) that are used by senders to
authenticate PIM protocol messages and are used by receivers to
authenticate received PIM protocol messages. This document does not
describe protocols for establishing SAs. It assumes that manual
configuration of SAs is performed, but it does not preclude the use
of a negotiation protocol such as the Internet Key Exchange [14] to
establish SAs.

IPsec [8] provides protection against replayed unicast and multicast
messages. The anti-replay option for IPsec SHOULD be enabled on all
SAs.

The following sections describe the SAs required to protect PIM
protocol messages.

6.3.1. Protecting Link-Local Multicast Messages

The network administrator defines an SA and SPI that are to be used
to authenticate all link-local PIM protocol messages (Hello,
Join/Prune, and Assert) on each link in a PIM domain.

IPsec [8] allows (but does not require) different Security Policy
Databases (SPD) for each router interface. If available, it may be
desirable to configure the Security Policy Database at a PIM router
such that all incoming and outgoing Join/Prune, Assert, and Hello
packets use a different SA for each incoming or outgoing interface.

6.3.2. Protecting Unicast Messages

   IPsec can also be used to provide data origin authentication and data
   integrity protection for the Register and Register-Stop unicast
   messages.

6.3.2.1. Register Messages

   The Security Policy Database at every PIM router is configured to
   select an SA to use when sending PIM Register packets to each
   rendezvous point.

   In the most general mode of operation, the Security Policy Database
   at each DR is configured to select a unique SA and SPI for traffic
   sent to each RP. This allows each DR to have a different
   authentication algorithm and key to talk to the RP. However, this
   creates a daunting key management and distribution problem for the
   network administrator. Therefore, it may be preferable in PIM
   domains where all Designated Routers are under a single
   administrative control that the same authentication algorithm
   parameters (including the key) be used for all Registered packets in
   a domain, regardless of who are the RP and the DR.

   In this "single shared key" mode of operation, the network
   administrator must choose an SPI for each DR that will be used to
   send it PIM protocol packets. The Security Policy Database at every
   DR is configured to select an SA (including the authentication
   algorithm, authentication parameters, and this SPI) when sending
   Register messages to this RP.

   By using a single authentication algorithm and associated parameters,
   the key distribution problem is simplified. Note, however, that this
   method has the property that, in order to change the authentication
   method or authentication key used, all routers in the domain must be
   updated.

6.3.2.2. Register-Stop Messages

   Similarly, the Security Policy Database at each Rendezvous Point
   should be configured to choose an SA to use when sending Register-
   Stop messages. Because Register-Stop messages are unicast to the
   destination DR, a different SA and a potentially unique SPI are
   required for each DR.

   In order to simplify the management problem, it may be acceptable to
   use the same authentication algorithm and authentication parameters,
   regardless of the sending RP and regardless of the destination DR.
   Although a unique SA is needed for each DR, the same authentication

      algorithm and authentication algorithm parameters (secret key) can be
      shared by all DRs and by all RPs.

6.4. Denial-of-Service Attacks

      There are a number of possible denial-of-service attacks against PIM
      that can be caused by generating false PIM protocol messages or even
      by generating data false traffic. Authenticating PIM protocol
      traffic prevents some, but not all, of these attacks. Three of the
      possible attacks include:

      - Sending packets to many different group addresses quickly can be a
         denial-of-service attack in and of itself. This will cause many
         register-encapsulated packets, loading the DR, the RP, and the
         routers between the DR and the RP.

      - Forging Join messages can cause a multicast tree to get set up. A
         large number of forged joins can consume router resources and
         result in denial of service.

      - Forging a (*,*,RP) join presents a possibility for a denial-of-
         service attack by causing all traffic in the domain to flow to the
         PMBR issuing the join. (*,*,RP) behavior is included here
         primarily for backwards compatibility with prior revisions of the
         spec. However, the implementation of (*,*,RP) and PMBR is
         optional. When using (*,*,RP), the security concerns should be
         carefully considered.

7. Acknowledgements

      PIM-SM was designed over many years by a large group of people,
      including ideas, comments, and corrections from Deborah Estrin, Dino
      Farinacci, Ahmed Helmy, David Thaler, Steve Deering, Van Jacobson, C.
      Liu, Puneet Sharma, Liming Wei, Tom Pusateri, Tony Ballardie, Scott
      Brim, Jon Crowcroft, Paul Francis, Joel Halpern, Horst Hodel, Polly
      Huang, Stephen Ostrowski, Lixia Zhang, Girish Chandranmenon, Brian
      Haberman, Hal Sandick, Mike Mroz, Garry Kump, Pavlin Radoslavov, Mike
      Davison, James Huang, Christopher Thomas Brown, and James Lingard.

      Thanks are due to the American Licorice Company, for its obscure but
      possibly essential role in the creation of this document.

8. Normative References

   [1]   Bradner, S., "Key words for use in RFCs to Indicate Requirement
          Levels", BCP 14, RFC 2119, March 1997.

   [2]   Cain, B., Deering, S., Kouvelas, I., Fenner, B., and A.
          Thyagarajan, "Internet Group Management Protocol, Version 3",
          RFC 3376, October 2002.

   [3]   Deering, S., "Host extensions for IP multicasting", STD 5, RFC
          1112, August 1989.

   [4]   Deering, S., Fenner, W., and B. Haberman, "Multicast Listener
          Discovery (MLD) for IPv6", RFC 2710, October 1999.

   [5]   Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6)
          Specification", RFC 2460, December 1998.

   [6]   Holbrook, H. and B. Cain, "Source-Specific Multicast for IP",
          RFC 4507, August 2006.

   [7]   IANA, "Address Family Numbers",
          <http://www.iana.org/assignments/address-family-numbers>.

   [8]   Kent, S. and K. Seo, "Security Architecture for the Internet
          Protocol", RFC 4301, December 2005.

   [9]   Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA
          Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

9. Informative References

   [10]   Bates, T., Rekhter, Y., Chandra, R., and D. Katz, "Multiprotocol
          Extensions for BGP-4", RFC 2858, June 2000.

   [11]   Bhaskar, N., Gall, A., Lingard, J., and S. Venaas, "Bootstrap
          Router (BSR) Mechanism for PIM Sparse Mode", Work in Progress,
          May 2006.

   [12]   Black, D., "Differentiated Services and Tunnels", RFC 2983,
          October 2000.

   [13]   Handley, M., Kouvelas, I., Speakman, T., and L. Vicisano, "Bi-
          directional Protocol Independent Multicast", Work in Progress,
          October 2005.

   [14]   Kaufman, C., "Internet Key Exchange (IKEv2) Protocol", RFC 4306,
          December 2005.

   [15] Savola, P., Lehtonen, R., and D. Meyer, "Protocol Independent
        Multicast - Sparse Mode (PIM-SM) Multicast Routing Security
        Issues and Enhancements", RFC 4609, August 2006.

   [16] Savola, P. and J. Lingard, "Last-hop Threats to Protocol
        Independent Multicast (PIM)", Work in Progress, January 2005.

   [17] Savola, P. and B. Haberman, "Embedding the Rendezvous Point (RP)
        Address in an IPv6 Multicast Address", RFC 3956, November 2004.

   [18] Thaler, D., "Interoperability Rules for Multicast Routing
        Protocols", RFC 2715, October 1999.

Appendix A. PIM Multicast Border Router Behavior

   In some cases, PIM-SM domains will interconnect with non-PIM
   multicast domains. In these cases, the border routers of the PIM
   domain speak PIM-SM on some interfaces and speak other multicast
   routing protocols on other interfaces. Such routers are termed PIM
   Multicast Border Routers (PMBRs). In general, RFC 2715 [18] provides
   rules for interoperability between different multicast routing
   protocols. In this appendix, we specify how PMBRs differ from
   regular PIM-SM routers.

   From the point of view of PIM-SM, a PMBR has two tasks:

   o To ensure that traffic from sources outside the PIM-SM domain
     reaches receivers inside the domain.

   o To ensure that traffic from sources inside the PIM-SM domain
     reaches receivers outside the domain.

   We note that multiple PIM-SM domains are sometimes connected together
   using protocols such as Multicast Source Discovery Protocol (MSDP),
   which provides information about active external sources, but does
   not follow RFC 2715. In such cases, the domains are not connected
   via PMBRs because Join(S,G) messages traverse the border between
   domains. A PMBR is required when no PIM messages can traverse the
   border.

A.1. Sources External to the PIM-SM Domain

   A PMBR needs to ensure that traffic from multicast sources external
   to the PIM-SM domain reaches receivers inside the domain. The PMBR
   will follow the rules in RFC 2715, such that traffic from external
   sources reaches the PMBR itself.

   According to RFC 2715, the PIM-SM component of the PMBR will receive
   an (S,G) Creation event when data from an (S,G) data packet from an
   external source first reaches the PMBR. If RPF_interface(S) is an
   interface in the PIM-SM domain, the packet cannot be originated into
   the PIM domain at this router, and the PIM-SM component of the PMBR
   will not process the packet. Otherwise, the PMBR will then act
   exactly as if it were the DR for this source (see Section 4.4.1),
   with the following modifications:

   o The Border-bit is set in all PIM Register messages sent for these
     sources.

   o DirectlyConnected(S) is treated as being TRUE for these sources.

   o The PIM-SM forwarding rule "iif == RPF_interface(S)" is relaxed to
     be TRUE if iif is any interface that is not part of the PIM-SM
     component of the PMBR (see Section 4.2).

A.2. Sources Internal to the PIM-SM Domain

   A PMBR needs to ensure that traffic from sources inside the PIM-SM
   domain reaches receivers outside the domain. Using terminology from
   RFC 2715, there are two possible scenarios for this:

   o Another component of the PMBR is a wildcard receiver. In this
     case, the PIM-SM component of the PMBR must ensure that traffic
     from all internal sources reaches the PMBR until it is informed
     otherwise.

     Note that certain profiles of PIM-SM (e.g., PIM-SSM, PIM-SM with
     Embedded RP) cannot interoperate with a neighboring wildcard
     receiver domain.

   o No other component of the PMBR is a wildcard receiver. In this
     case the PMBR will receive explicit information as to which groups
     or (source,group) pairs the external domains wish to receive.

   In the former case, the PMBR will need to send a Join(*,*,RP) to all
   the active RPs in the PIM-SM domain. It may also send a Join(*,*,RP)
   to all the candidate RPs in the PIM-SM domain. This will cause all
   traffic in the domain to reach the PMBR. The PMBR may then act as if
   it were a DR with directly connected receivers and trigger the
   transition to a shortest path tree (see Section 4.2.1).

   In the latter case, the PMBR will not need to send Join(*,*,RP)
   messages. However, the PMBR will still need to act as a DR with
   directly connected receivers on behalf of the external receivers in
   terms of being able to switch to the shortest-path tree for
   internally-reached sources.

   According to RFC 2715, the PIM-SM component of the PMBR may receive a
   number of alerts generated by events in the external routing
   components. To implement the above behavior, one reasonable way to
   map these alerts into PIM-SM state is as follows:

   o When a PIM-SM component receives an (S,G) Prune alert, it sets
     local_receiver_include(S,G,I) to FALSE for the discard interface.

   o When a PIM-SM component receives a (*,G) Prune alert, it sets
     local_receiver_include(*,G,I) to FALSE for the discard interface.

o When a PIM-SM component receives an (S,G) Join alert, it sets
  local_receiver_include(S,G,I) to TRUE for the discard interface.

o When a PIM-SM component receives a (*,G) Join alert, it sets
  local_receiver_include(*,G,I) to TRUE for the discard interface.

o When a PIM-SM component receives a (*,*) Join alert, it sets
  DownstreamJPState(*,*,RP,I) to Join state on the discard interface
  for all RPs in the PIM-SM domain.

o When a PIM-SM component receives a (*,*) Prune alert, it sets
  DownstreamJPState(*,*,RP,I) to NoInfo state on the discard
  interface for all RPs in the PIM-SM domain.

We refer above to the discard interface because the macros and state
machines are interface specific, but we need to have PIM state that
is not associated with any actual PIM-SM interface. Implementers are
free to implement this in any reasonable manner.

Note that these state changes will then cause additional PIM-SM state
machine transitions in the normal way.

These rules are, however, not sufficient to allow pruning off the
(*,*,RP) tree. Some additional rules provide guidance as to one way
this may be done:

o If the PMBR has joined on the (*,*,RP) tree, then it should set
  DownstreamJPState(*,G,I) to JOIN on the discard interface for all
  active groups.

o If the router receives a (S,G) prune alert, it will need to set
  DownstreamJPState(S,G,rpt,I) to PRUNE on the discard interface.

o If the router receives a (*,G) prune alert, it will need to set
  DownstreamJPState(S,G,rpt,I) to PRUNE on the discard interface for
  all active sources sending to G.

The rationale for this is that there is no way in PIM-SM to prune
traffic off the (*,*,RP) tree, except by Joining the (*,G) tree and
then pruning each source individually.

Appendix B. Index

Authors' Addresses

   Bill Fenner
   AT&T Labs - Research
   1 River Oaks Place
   San Jose, CA 95134

   EMail: fenner@research.att.com

   Mark Handley
   Department of Computer Science
   University College London
   Gower Street
   London WC1E 6BT
   United Kingdom

   EMail: M.Handley@cs.ucl.ac.uk

   Hugh Holbrook
   Arastra, Inc.
   P.O. Box 10905
   Palo Alto, CA 94303

   EMail: holbrook@arastra.com

   Isidor Kouvelas
   Cisco Systems
   170 W. Tasman Drive
   San Jose, CA 95134

   EMail: kouvelas@cisco.com

Full Copyright Statement

Intellectual Property

Acknowledgement