



# Treck IPv6 Supplement



## Complete Internet Solutions

**Technical Support:**

5041 Lamart Drive #240 Riverside, California 92507  
Phone: (909) 787-7056 Fax: (909) 787-8803

**Corporate Headquarters:**

Treck, Inc. 431 Ohio Pike Suite 210 North Cincinnati, Ohio 45255  
Phone: (513) 528-5732 Fax: (513) 528-5740

# Contents

<b>Related Documents .....</b>	<b>4</b>
<b>1 Treck IPv6 .....</b>	<b>5</b>
<b>2 Description .....</b>	<b>5</b>
2.1 Modes of Operation .....	5
2.2 Reuse of Existing IPv4 Data Structures .....	5
2.3 Naming Standards for New APIs .....	5
2.4 New Public APIs .....	6
2.5 Changes to Existing Public APIs .....	7
2.5.1 Deprecated APIs .....	7
2.5.2 APIs Not Yet Deprecated .....	8
2.6 Changes to Existing Internal APIs .....	9
2.6.1 128-bit IP Addresses .....	9
2.6.2 Timer Redesign .....	9
2.6.3 Prefix Length versus Netmask .....	9
2.7 Scope ID .....	10
2.8 IPv6 Neighbor Cache versus ARP Cache .....	10
<b>3 Limitations .....</b>	<b>11</b>
3.1 Router .....	11
3.2 DNS Resolver API .....	11
<b>4 Function Reference .....</b>	<b>12</b>
4.1 BSD Socket APIs .....	12
4.1.1 accept .....	12
4.1.2 bind .....	12
4.1.3 connect .....	12
4.1.4 freeaddrinfo .....	12
4.1.5 gai_strerror .....	13
4.1.6 getaddrinfo .....	14
4.1.7 getnameinfo .....	16
4.1.8 getpeername .....	17
4.1.9 getsockname .....	17
4.1.10 getsockopt .....	17
4.1.11 if_indextoname .....	18
4.1.12 if_nametoindex .....	19
4.1.13 inet_ntop .....	20
4.1.14 inet_pton .....	21
4.1.14.1 <i>inet_pton Example</i> .....	22
4.1.15 recvfrom .....	24
4.1.16 sendto .....	24
4.1.17 setsockopt .....	24
4.1.18 socket .....	24
4.2 Socket Extension Calls .....	25
4.2.1 tf6AddressToIpv4Compat .....	25
4.2.2 tf6AddressToIpv4Mapped .....	26
4.2.3 tf6GetLocalIpAddress .....	27
4.2.4 tf6SockaddrSetScopeId .....	29
4.2.4.1 <i>tf6SockaddrSetScopeId Example</i> .....	30
4.3 Treck Initialization Functions .....	31
4.3.1 tfInitTreckOptions .....	31
4.3.2 tfSetTreckOptions .....	31
4.4 Device / Interface API .....	32
4.4.1 tf4NetmaskToPrefixLen .....	32
4.4.2 tf6Eui64GetInterfaceId .....	33

4.4.3	tf6Eui64SetInterfaceId .....	34
4.4.4	tf6InterfaceSetPhysAddr .....	36
4.4.5	tf6InterfaceSetSiteId .....	37
4.4.6	tf6SetMcastInterface .....	38
4.4.7	tf6SetRandomInterfaceId .....	39
4.4.8	tfCheckOpenInterface .....	40
4.4.8.1	<i>tfCheckOpenInterface Example</i> .....	41
4.4.9	tfInterfaceSetOptions .....	42
4.4.10	tfNgConfigInterface .....	43
4.4.11	tfNgGetIpAddress .....	46
4.4.12	tfNgGetPrefixLen .....	47
4.4.13	tfNgOpenInterface .....	48
4.4.13.1	<i>tfNgOpenInterface Example</i> .....	50
4.4.14	tfNgUnConfigInterface .....	52
4.5	ARP/ Routing Table API .....	53
4.5.1	tf6AddDefaultGatewayTunnel .....	53
4.5.2	tf6DelDefaultGatewayTunnel .....	54
4.5.3	tf6GetDefaultGatewayTunnel .....	55
4.5.4	tf6GetPathMtu .....	55
4.5.5	tf6RegisterIpForwCB .....	56
4.5.6	tfNgAddArpEntry .....	56
4.5.7	tfNgAddStaticRoute .....	57
4.5.8	tfNgDelArpEntryByIpAddr .....	57
4.5.9	tfNgDelArpEntryByPhysAddr .....	58
4.5.10	tfNgDelStaticRoute .....	59
4.5.11	tfNgDisablePathMtuDisc .....	60
4.5.12	tfNgGetArpEntryByIpAddr .....	61
4.5.13	tfNgGetArpEntryByPhysAddr .....	62
4.5.14	tfSetReachable .....	63
4.6	PPP Link Layer API .....	64
4.6.1	Using PPP in dual IPv4/IPv6 mode .....	64
4.6.1.1	<i>PPP Dual IPv4/Ipv6 Usage Example</i> .....	64
4.6.2	tfNgGetPt2PtPeerIpAddress .....	65
4.6.3	tfNgSetPt2PtPeerIpAddress .....	65
4.6.4	tfPppSetOption .....	66
4.6.5	tfUseAsyncPpp .....	67
4.6.6	tfUseAsyncServerPpp .....	69
<b>5</b>	<b>Application Reference .....</b>	<b>71</b>
5.1.1	tfNgDnsSetServer .....	71
5.1.2	tfNgFtpConnect .....	72
5.1.3	tfNgFtpdUserStart .....	73
5.1.4	tfNgPingOpenStart .....	74
5.1.4.1	<i>tfNgPingOpenStart Example</i> .....	75
5.1.5	tfNgTeldOpened .....	76
<b>6</b>	<b>Structure Reference .....</b>	<b>77</b>
6.1.1	addrinfo .....	77
6.1.2	in6_addr .....	78
6.1.3	ipv6_mreq .....	79
6.1.4	sockaddr_in6 .....	80
6.1.5	sockaddr_storage .....	81
6.1.6	tt6LocalIpAddressCursor .....	82
<b>7</b>	<b>Macros .....</b>	<b>83</b>
7.1	Performance Macros .....	83
7.2	BSD Socket API Macros .....	83
7.3	Socket Extension Macros .....	86

## Related Documents

Document ID	Document Name
[RFC2373]	RFC-2373: IP Version 6 Addressing Architecture
[RFC2374]	RFC-2374: An IPv6 Aggregatable Global Unicast Address Format
[RFC2375]	RFC-2375: IPv6 Multicast Address Assignments
[RFC2526]	RFC-2526: Reserved IPv6 Subnet Anycast Addresses
[RFC2460]	RFC-2460: Internet Protocol, Version 6 (IPv6) Specification
[RFC2463]	RFC-2463: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) specification
[RFC2461]	RFC-2461: Neighbor Discovery for IP Version 6 (IPv6)
[RFC2462]	RFC-2462: IPv6 Stateless Address Auto-configuration
[RFC2473]	RFC-2473: Generic Packet Tunneling in IPv6 Specification
[RFC2710]	RFC-2710: Multicast Listener Discovery (MLD) for IPv6
[RFC1981]	RFC-1981: Path MTU Discovery for IP version 6
[RFC2185]	RFC-2185: Routing Aspects Of IPv6 Transition
[RFC2893]	RFC-2893: Transition Mechanisms for IPv6 Hosts and Routers
[RFC2464]	RFC-2464: Transmission of IPv6 Packets over Ethernet Networks
[RFC2472]	RFC-2472: IP Version 6 over PPP
[RFC2553]	RFC-2553: Basic Socket Interface Extensions for IPv6
[RFC2292]	RFC-2292: Advanced Sockets API for IPv6
[RFC2711]	RFC-2711: IPv6 Router Alert Option
[RFC3056]	RFC-3056: Connection of IPv6 Domains via IPv4 Clouds
[RFC2080]	RFC-2080: RIPng for IPv6
[RFC2474]	RFC-2474: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers
[CASE_FOR_IPV6]	draft-iab-case-for-ipv6-06.txt: The Case for IPv6
[ASSIGNED]	Reynolds, J. and J. Postel, "ASSIGNED NUMBERS", STD 2, RFC-1700, October 1994. See also: <a href="http://www.iana.org/numbers.html">http://www.iana.org/numbers.html</a>
[RFC2465]	RFC-2465: Management Information Base for IP Version 6: Textual Conventions and General Group
[RFC2466]	RFC-2466: Management Information Base for IP Version 6: ICMPv6 Group
[RFC2471]	RFC-2471: IPv6 Testing Address Allocation
[RFC2851]	RFC-2851: Textual Conventions for Internet Network Addresses
[MOBILE_IPV6]	draft-ietf-mobileip-ipv6-13.txt: Mobility Support in IPv6
[STEVENS2]	"TCP/IP Illustrated, Volume 2 The Implementation", by Gary R. Wright and W. Richard Stevens, ISBN 0-201-63354-X
[RFC2401]	RFC-2401: Security Architecture for the Internet Protocol
[RFC2402]	RFC-2402: IP Authentication Header
[RFC2406]	RFC-2406: IP Encapsulating Security Payload (ESP)
[HEAPSPEC]	Treck TCP/IP Heap and Buffer Specification, v1.5. Filename: heapspec.doc
[IPV6REQ]	The master requirements document for phase 1 IPv6. Filename: ipv6Requirements.doc
[RFC3041]	RFC-3041: Privacy Extensions for Stateless Address Autoconfiguration in IPv6
[DEFADDR_07]	draft-ietf-ipv6-default-addr-select-07.txt (IETF draft RFC) Default Address Selection for IPv6

# 1 Treck IPv6

This manual is intended as supplemental documentation for the Treck TCP/IP User's Manual.

## 2 Description

### 2.1 Modes of Operation

Our dual IP layer stack will support three different modes of operation:

1. IPv4-only
2. IPv6-only
3. Dual-mode IPv6 with IPv4 (also referred to as “dual IP layer”)

This is done in a way (i.e. conditional compilation using the macros `TM_USE_IPV4` and `TM_USE_IPV6`) that minimizes codespace and dataspace requirements for customers who prefer to run the stack in IPv4-only or IPv6-only modes. The default mode of operation is IPv4-only to be compatible with our current product. We have implemented IPv6 functionality as an option so that if a customer does not buy the IPv6 option, they will not receive the files required to build IPv6 support. Therefore, they will get linker errors if they try to use IPv6. If the user disables both IPv4 and IPv6 support (an invalid configuration) an appropriate error message will be generated at compile time using the `#error` preprocessor directive. Also, when `TM_USE_IPV6` is `#define'd`, `TM_SINGLE_INTERFACE_HOME` cannot be `#define'd` because IPv6 interfaces are multi-homed. When both `TM_USE_IPV6` and `TM_SINGLE_INTERFACE_HOME` are `#define'd` (an invalid configuration) an appropriate error message will be generated at compile time using the `#error` preprocessing directive. The performance macro `TM_SINGLE_INTERFACE` can be `#define'd` when `TM_USE_IPV6` is enabled to reduce codesize. However, you will be restricted to using a single interface.

### 2.2 Reuse of Existing IPv4 Data Structures

There are many similarities between IPv6 and IPv4 in the sense that IPv6 keeps the best parts of IPv4. Because of this, we try to reuse existing IPv4 internal data structures by making simple changes and extensions to them where applicable. For example, both the Patricia routing tree (which also comprises ARP cache functionality) and the socket tree can be adapted to support IPv6 by expanding the size of the IP address to 128 bits when operating in dual-mode IPv6 with IPv4 (in this case, IPv4-only addresses are stored internally as 128-bit IPv4-mapped IPv6 addresses). As another example, the existing IPv4 interface structure (i.e. `ttDeviceEntry`) should be straightforward to adapt for IPv6 use by adding new IPv6-specific fields such as the IPv6 interface ID (i.e. `dev6InterfaceId`), a multi-homed IPv6 address list containing both manually configured as well as stateless auto-configured IPv6 addresses (i.e. `dev6IpAddrArray`, `dev6PrefixLenArray`), a list of IPv6 multicast addresses that the interface is listening on (i.e. `dev6MldPtr`), etc.

To make it clear that specific fields in these shared internal data structures are either IPv4-only or IPv6-only, these fields will be bracketed by the appropriate `#ifdef's` for selecting IPv4 or IPv6 functionality, which are `TM_USE_IPV4` and `TM_USE_IPV6` respectively. An example of a field that is IPv4-only is the existing multi-homed IPv4 address list (i.e. `ttDeviceEntry.devIpAddrArray`, `ttDeviceEntry.devNetMaskArray`).

In summary, changes to existing IPv4 functionality shall be kept to a minimum, and specifically we do not want any of our changes for IPv6 to increase codespace and dataspace usage when the stack is used in IPv4-only mode.

### 2.3 Naming Standards for New APIs

Any new APIs (both public and internal) which are IPv6-only have names starting with the prefix “tf6” unless the RFC requires otherwise. Any new APIs which support both IPv4 and IPv6 have names starting with the prefix “tf”, except for when the new API replaces an existing IPv4-only API. In that case, the name starts with the prefix “tfNg” as described in the next section “*Changes to Existing Public APIs*”.

Similar naming conventions are used for macros, type names, and global variables. For example, “TM\_6\_” is used as the prefix for IPv6-only macros, “TM\_” for macros that support both IPv4 and IPv6 except for in the case where it replaces an existing IPv4-only macro in which case the prefix is “TM\_NG\_”, etc.

IPv6-specific fields added to existing IPv4 data structures have “6” added at the end of the fieldname prefix if there is a prefix, for example: `ttDeviceEntry.dev6IpAddrArray` is the list of IPv6 unicast addresses configured on the interface. In a similar manner, “4” may be used at the end of the prefix to indicate IPv4-only, for example: `ttRtRadixHeadEntry.rth4DefaultGateway` is the routing entry for the IPv4 default gateway.

## 2.4 New Public APIs

[RFC2553] specifies new public BSD socket APIs. Some of these new APIs we won't implement in phase 1 IPv6, because they require the DNS resolver API to be updated for IPv6 (targeted for phase 2 IPv6), and these APIs are: `freeaddrinfo`, `freehostent`, `gai_strerror`, `getaddrinfo`, `getipnodebyname`, `getipnodebyaddr` and `getnameinfo`. Additionally, we are not implementing `if_nameindex` and `if_freenameindex`, because they involve dynamic memory allocation that is not appropriate for embedded systems, and equivalent functionality can be achieved by calling other APIs (specifically, by calling `if_indextoname` with sequentially increasing values for interface index until the function returns NULL). The other new BSD socket APIs specified in [RFC2553] we will implement, which include `if_nametoindex`, `if_indextoname`, `inet_pton` and `inet_ntop`. ([IPV6REQ].R2.18:10)

ICMPv6 Neighbor Discovery specifies a means to perform address resolution, which works for other link layers besides Ethernet (i.e. null link-layer). All that is required to use ICMPv6 address resolution with an interface is to configure the link-layer address of that interface. This can be accomplished by calling the new public API `tf6InterfaceSetPhysAddr` from the device driver open function. Note that if `tf6InterfaceSetPhysAddr` is used instead of `drvGetPhyAddrFuncPtr` to set the link-layer address, then `tf6Eui64SetInterfaceId` should also be called from the device driver open function to set the interface ID.

Because of how IPv6 stateless address auto-configuration works, IPv6 needs access to the interface ID. The interface ID is always 64 bits long ([RFC2373].R2.4:20), and is used as the low-order 64 bits of a stateless auto-configured IPv6 address. A new public API `tf6Eui64SetInterfaceId` enables the user to configure the interface ID associated with a specific interface, however it should not be called when the interface link-layer type is Ethernet, since in that case the interface ID is automatically derived from the IEEE 48-bit MAC address provided by the user's `driverGetPhysicalAddress` function (refer to the `drvGetPhysAddrFuncPtr` parameter of the function `tfAddInterface`). If the interface link-layer type is null link-layer, then `tf6Eui64SetInterfaceId` should be called by the user from the device driver open function. If the interface ID is not set before the interface is opened, then IPv6 stateless address auto-configuration will not be performed, which may result in the IPv6 interface not being opened depending on what IPv6 addresses have been manually configured on the interface (i.e. at least one link-local IPv6 address is required). ([RFC2464].R4:20, [RFC2373].R2.1:10, [RFC2373].R2.4:20)

Because of how scoped IPv6 addresses work, specifically site-local unicast IPv6 addresses, IPv6 needs access to the site identifier. A new API `tf6InterfaceSetSiteId` enables the user to configure the site identifier associated with a specific interface. Typically, all interfaces in the node will belong to the same site, therefore we default the site identifier associated with each interface to 0, and do not require the user to call `tf6InterfaceSetSiteId` in this case. Refer to section 2.2.1 "Scope ID" for more information on site identifiers.

Since in some cases the user may need to set the scope ID of a local scope IPv6 address to associate it with the specific interface they want to use it with (for example, sending packets to a link-local scope unicast IPv6 address requires specifying the outgoing interface), we provide the API `tf6SockaddrSetScopeId` to set the appropriate scope ID to use. This API can be used either with the BSD socket APIs or with Treck APIs (i.e. `tf6SockaddrSetScopeId` can be used to assign the scope ID to `sockaddr_in6.sin6_scope_id`).

IPv6 address prefixes encode the scope of the address, which in the case of a unicast IPv6 address is link-local, site-local or global. Link-local, site-local and global scope unicast addresses are automatically assigned to an interface using stateless address auto-configuration. Additional IPv6 unicast addresses may be manually configured on the interface. Addresses may become deprecated, in which case they should not be used in any new communications. Because there will be more than one local IPv6 address associated with an IPv6 interface, and since there are various criteria (i.e. local versus global scope, preferred versus deprecated status, etc.) governing which are the appropriate local IP addresses to use for communication with a specific destination IP address, we provide a new API `tf6GetLocalIpAddress` which enables the application to select an appropriate local IP address. ([RFC2462].R4.1:10)

To make it easier for the user to build addresses of type `sockaddr_storage`, we provide the new APIs `tf6AddressToIpv4Compat` and `tf6AddressToIpv4Mapped`.

A new API `tfCheckOpenInterface` is provided which enables the user to check the status of either IPv4 or IPv6 protocol operation on a specified interface.

A new API `tf6SetMcastInterface` is provided which enables the user to set the default interface to use to send IPv6 multicast packets. The application can set the `IPV6_MULTICAST_IF` option on the socket to override this default interface on a per-socket basis.

`tfRegisterIpForwCB` functionality needs to be provided for IPv6. A new API `tf6RegisterIpForwCB` is provided for this purpose.

`tf4NetmaskToPrefixLen` is provided to facilitate porting of existing code to the `tfNg` APIs.

A new API `tfSetReachable` is provided so that applications can indicate that a connection is making forward progress. ([RFC2461].R7.3.1:10, [RFC2461].R7.3.1:20, [RFC2461].R7.3.1:30, [RFC2461].R7.3.1:40)

## 2.5 Changes to Existing Public APIs

Because we must not break any existing code that a customer has written, we have made no changes to the public API that are not backwards compatible. Generally, our intent is similar to that described in [RFC2553], which is to provide backwards-compatibility for changes to existing public APIs and when new public APIs are defined, to have these new APIs support both IPv4 and IPv6 where it makes sense.

From the standpoint of the existing public API, the primary impact that IPv6 has is due to the expansion of the size of an IP address from 32 bits (IPv4) to 128 bits (IPv6). Unfortunately, the existing type used in the public API to represent an IP address (i.e. `ttUserIpAddress`) does not easily support expansion to 128 bits because it is an integral type rather than a pointer type. Therefore, we are using a new BSD type (i.e. `sockaddr_storage`) to represent an IP address and new APIs that are passed a pointer to this type.

Because the existing APIs do not support IPv6 addresses, we have deprecated these existing IPv4-only APIs and recommend that users convert their code to use the new APIs that support both IPv4 and IPv6 addresses. These new APIs that replace existing IPv4-only APIs have names starting with the prefix “`tfNg`”, which is an abbreviation for *Treck Function Next Generation*. The new APIs call the deprecated APIs (as needed).

The existing BSD socket API has been updated in accordance with [RFC2553]. This preserves backwards compatibility with existing applications that use the old version of this API. ([IPV6REQ].R2.18:10)

A new option has been added to `tfSetTreckOptions` that enables the application to prevent any new IPv6 communication from using a deprecated local IPv6 address. ([RFC2462].R5.5.4:60)

A new option has been added to `tfInterfaceSetOptions` that enables the application to control per interface how many consecutive Neighbor Solicitation messages are sent while performing Duplicate Address Detection on a tentative IPv6 address. ([RFC2462].R5.1:10, [RFC2462].R5.1:20)

### 2.5.1 Deprecated APIs

The following APIs are deprecated because of the new address type `sockaddr_storage` supporting both IPv4 and IPv6 addresses, and are replaced by similarly named *Next Generation* (i.e. “`tfNg`”) APIs:

- `tfAddArpEntry`
- `tfAddStaticRoute`
- `tfConfigInterface`
- `tfDelArpEntryByIpAddr`
- `tfDelArpEntryByPhysAddr`
- `tfDelStaticRoute`
- `tfFtpConnect`
- `tfFtpdUserStart`
- `tfGetArpEntryByPhysAddr`
- `tfGetArpEntryByIpAddr`
- `tfGetIpAddress`
- `tfGetNetMask`
- `tfGetPppPeerIpAddress`
- `tfNgDnsSetServer`
- `tfOpenInterface`
- `tfPingOpenStart`
- `tfSetPppPeerIpAddress`
- `tfTeldOpened`
- `tfUnConfigInterface`

This requires a change to user documentation to indicate that these APIs are deprecated and to identify the Next Generation APIs which should be called in their place.

### 2.5.2 APIs Not Yet Deprecated

Because we have not implemented the equivalent of proxy-ARP in phase 1 IPv6 (targeted for phase 3 IPv6, router), we will not yet deprecate **tfAddInterfaceMhomeAddress**, **tfAddProxyArpEntry** or **tfDelProxyArpEntry**.

Because we have not implemented Path MTU Discovery in phase 1 IPv6 (targeted for phase 2 IPv6), we will not yet deprecate **tfDisablePathMtuDisc**.

Some existing public APIs that take a parameter of type *ttUserIpAddress* have not been deprecated because they are only used with IPv4. The default gateway functions **tfAddDefaultGateway**, **tfDelDefaultGateway** and **tfGetDefaultGateway** are not deprecated since these functions only make sense to use with IPv4. With IPv6, because of its plug-and-play capabilities (specifically, router discovery), there is no need to have an API for setting the default IPv6 gateway.

For testing IPv6 with a router that does not support sending Router Advertisement messages (i.e. testing IPv6 forwarding), the user can effectively override the IPv6 default router list by adding a static route having a prefix length of 3 and a destination IPv6 address of 2000::0 which matches aggregatable global unicast IPv6 addresses.

**tfGetBroadcastAddress** is not deprecated because IPv6 does not support broadcast.

The DNS APIs we are not looking at for phase 1 IPv6; these will likely be deprecated later when we update the DNS resolver API for IPv6, but not now. This includes **tfGetPppDnsIpAddress**.

BOOTP and DHCP are protocols specific to IPv4, therefore these APIs will not be deprecated. There will be a DHCPv6 stateful configuration protocol for IPv6 though currently may have draft status. Additionally, it is out of the scope for Phase 1 IPv6.

**tfUseCollisionDetection**, **tfUserStartArpSend** and **tfCancelCollisionDetection** have not been deprecated since they are used by AutoIP and are specific to IPv4. IPv6 provides similar functionality via Duplicate Address Detection, which is part of IPv6 stateless address auto-configuration (refer to [RFC2462]), however the only API that IPv6 stateless address auto-configuration requires is a means for the application to set the interface ID (refer to **tf6Eui48SetInterfaceId**, **tf6Eui64SetInterfaceId**).

The scattered send APIs **tfIpScatteredSend** and **tfSocketScatteredSendTo** are out of scope for phase 1 IPv6.

**tfFinishOpenInterface** does not make sense to deprecate. Every IPv6 address configured on the interface, regardless of whether it was manually configured, obtained via stateful configuration (i.e. DHCPv6) or stateless auto-configuration, must go through the process of Duplicate Address Detection ([RFC2462].R5.4:10) before **tfDeviceStart** can be called to complete configuration of the IPv6 address on the interface. **tfFinishOpenInterface** is a means for the user to call **tfDeviceStart**, but since we can't allow them to call **tfDeviceStart** until Duplicate Address Detection has completed successfully, it makes no sense to provide this API for IPv6.

## 2.6 Changes to Existing Internal APIs

### 2.6.1 128-bit IP Addresses

When the stack is run in dual IP layer mode, IPv4-only IP addresses are represented internally as 128-bit IPv4-mapped IPv6 addresses so that we can share existing internal data structures (and associated code) such as the socket tree and the Patricia routing tree between IPv4 and IPv6 implementations. This is accomplished as follows: when `TM_USE_IPV6` is `#define'd`, `ttIpAddress` (i.e. the internal type used for IPv4 addresses) is `#define'd` to be `tt6IpAddress`. All internal APIs and data structures that reference the type `ttIpAddress` are impacted by this change (i.e. `ttSocketEntry`, `ttRtCacheEntry`, `ttSocketLookup`, etc.)

Internal API optimization, conserve stack space:

When `TM_USE_IPV6` is `#define'd`, all parameters of type `ttIpAddress` are passed as `tt6IpAddress`, which means that each consumes 128 bits of stack space. These APIs will be conditionally changed via the `TM_USE_IPV6` macro to instead take parameters of type `ttIpAddressPtr`.

### 2.6.2 Timer Redesign

Because there are a number of new timer-related requirements for ICMPv6, IPv6 address prefixes to implement preferred lifetimes, and valid lifetimes of IPv6 addresses derived from those prefixes, we will optimize our existing timer implementation which currently takes a brute-force approach of updating a countdown for each timer in the timer active queue whenever a call to `ttTimerExecute` occurs.

Specifically, we will keep all active timers in a singly-linked-list, ordered by the time they are due to expire, with the next timer to expire at the head of the list. Additionally, the timer will not store a countdown that needs to be updated every time `ttTimerExecute` is called, but instead will store the tick count (i.e. `tvTime`) latched at the time when the timer was started, and this start time is used to compute the elapsed time (i.e. elapsed time is the modulo 32-bit unsigned difference between the current time and the start time). This enables `ttTimerExecute` to compute the elapsed time for only those active timers that have expired up to and including the first active timer that hasn't expired, but excluding the remaining timers. When there is a large number of active timers, and the majority of them are not due to expire, then this algorithm will be more efficient.

### 2.6.3 Prefix Length versus Netmask

Obviously, it is very inefficient to pass around and store a 128-bit netmask for IPv6 when instead you can use an 8-bit prefix length. Prefix length can take on values ranging from 0 to 128, and it indicates the number of most significant bits set in the netmask.

Dual IP layer mode optimization, IPv4 prefix length stored on interface:

Change IPv4 to use prefix length instead of netmask. When the stack is running in dual IP layer mode, an IPv4 netmask (i.e. `ttDeviceEntry.devNetMaskArray`) is stored for each multi-homed IPv4 address on the interface using 128 bits (i.e. `ttIpAddress`, which is `#define'd` to be `tt6IpAddress` in this case). This is changed to instead store the 8-bit prefix length (i.e. `ttDeviceEntry.dev4PrefixLenArray`).

## 2.7 Scope ID

The addition of scoped unicast addressing to IPv6 complicates routing as well as the design of internal data structures which store IPv6 addresses. For example, you can have the same link-local scope unicast IPv6 address assigned to different nodes on different links, and therefore reachable through different interfaces. If the user tries to send a packet to this link-local scope IPv6 address, which interface should the packet go out on? Remember that in this example we are addressing different nodes and not the same node through different interfaces. Therefore, the choice of which outgoing interface to use matters.

To work around this issue, IETF added the field *sin6\_scope\_id* to the *sockaddr\_in6* structure that is part of the BSD socket API for IPv6. According to [RFC2553], this field contains an interface index (i.e. *ttDeviceEntry.devIndex*, note that this is a 1-based index) if the scope of the unicast IPv6 address is link-local, and a site identifier (i.e. *ttDeviceEntry.dev6SiteId*) if the scope is site-local. The use of the field *sockaddr\_in6.sin6\_scope\_id* is implementation dependent, as long as the value chosen for scope ID is unique at the appropriate scope level so that the combination of scope ID and local scope unicast IPv6 address is unique internal to the node. The user can call **tf6SockaddrSetScopeId** to assign the correct value to *sockaddr\_in6.sin6\_scope\_id*.

A scope identifier may be 0, which refers to the default scope when a valid scope identifier is not known (refer to [RFC2851]). We initialize the site identifier of each interface to 0; the user can call the API **tf6InterfaceSetSiteId** to set this to a valid non-zero scope identifier for site-local unicast IPv6 addresses associated with a specific interface.

## 2.8 IPv6 Neighbor Cache versus ARP Cache

In our current IPv4 implementation, the ARP Cache is part of the Patricia routing tree. IPv6 will use the ARP Cache to store Neighbor Cache information, however first the ARP Cache entry needs to be modified so that it can handle variable-length link-layer addresses.

## 3 Limitations

### 3.1 Router

Treck, Inc. is only doing a host implementation of IPv6 in phase 1 IPv6.

### 3.2 DNS Resolver API

1. A6 queries are not implemented:  
We are not implementing support for A6 name lookups (part of [RFC2874]) in phase 2 IPv6, however we are implementing support for [RFC1886] (AAAA). A6 name lookups are significantly different from our existing IPv4 DNS Resolver implementation; specifically, several round trips may be needed to collect a complete DNS record when using A6. Currently, it is unclear as to if or when [RFC1886] will be deprecated. Also, [RFC2874] specifies a transition mechanism that IPv6 DNS servers can use to populate their AAAA records with the information in their A6 records, so that they can continue to support AAAA name lookups even after they've converted over to A6.
2. The general versions of **getaddrinfo** and **getnameinfo**, described in [BASIC\_API\_07], both support a series of features based on service names and port numbers. The goal of this is primarily to combine the functionality of a number of API calls (**getservbyname**, **gethostbyname**, etc) into a single, versatile, protocol-independent function. However, our implementation is focused on embedded systems, which typically do not have the capability (or desire) to do a **getservbyname** (i.e. no filesystem, no /etc/services file, etc). Because of this, none of the service and port number related functions of **getaddrinfo** and **getnameinfo** are implemented. This means that the *servname* parameters are not used, and the following flags are not defined and may not be used:  
AI\_PASSIVE, AI\_NUMERICSERV, NI\_NOFQDN, NI\_NUMERICSERV, NI\_NUMERICSCOPE and NI\_DGRAM.
3. There is currently no API for retrieving IPv6 mail host (MX) records. The DNS code is capable of generating and receiving IPv6 MX records, however they are not stored, since there is not yet an API for the user to retrieve this information. This is a rarely used functionality - to the best of my knowledge, none of our customers currently retrieve IPv4 MX records, even though the capability is there.

## 4 Function Reference

### 4.1 BSD Socket APIs

#### 4.1.1 `accept`

This BSD socket API is updated to add support for the IPv6 address type *sockaddr\_in6*. The function prototype is unchanged.

When applications use PF\_INET6 sockets to accept TCP connections from IPv4 nodes or receive UDP packets from IPv4 nodes, the system returns the peer's address to the application in the **accept**, **recvfrom**, or **getpeername** call using a *sockaddr\_in6* structure with the peer's IPv4 address encoded as an IPv4-mapped IPv6 address. ([IPV6REQ].R2.18:10, [RFC2553].R3.3:10)

#### 4.1.2 `bind`

This BSD socket API is updated to add support for the IPv6 address type *sockaddr\_in6*, including setting the flow information (i.e. *sockaddr\_in6.sin6\_flowinfo*) associated with the socket. The function prototype is unchanged.

([IPV6REQ].R2.18:10)

#### 4.1.3 `connect`

This BSD socket API is updated to add support for the IPv6 address type *sockaddr\_in6*, including setting the flow information (i.e. *sockaddr\_in6.sin6\_flowinfo*) associated with the socket. The function prototype is unchanged.

Applications may use PF\_INET6 sockets to open TCP connections to IPv4 nodes, or send UDP packets to IPv4 nodes, by simply encoding the destination's IPv4 address as an IPv4-mapped IPv6 address, and passing that address, within a *sockaddr\_in6* structure, in the **connect** or **sendto** call.

([IPV6REQ].R2.18:10)

#### 4.1.4 `freeaddrinfo`

```
void                freeaddrinfo
(
  struct addrinfo *  addrInfoPtr
);
```

#### Function Description

Frees memory used by address information structure chain returned by **getaddrinfo**.

#### Parameters

Parameter	Description
<i>addrInfoPtr</i>	Pointer to first address info structure to be freed.

#### Returns

None

#### 4.1.5 `gai_strerror`

```
char *      gai_strerror  
(  
int        errorCode  
);
```

##### **Function Description**

Returns a string describing the error code, returned from `getaddrinfo` or `getnameinfo`.

##### **Parameters**

<b>Parameter</b>	<b>Description</b>
<i>errorCode</i>	EAI_XXX error code.

##### **Returns**

Pointer to string buffer describing the error.

### 4.1.6 getaddrinfo

```

int          getaddrinfo
(
const char *  nodeName,
const char *  serviceName,
const struct addrinfo * hintsPtr,
struct addrinfo ** resPtrPtr
);

```

#### Function Description

Translates a node-name (hostname) to an address. Similar to **gethostbyname**, used for both IPv4 and IPv6. Retrieving a service name or port number is not supported in our implementation of **getaddrinfo**, so `serviceName` must be set to `NULL`.

**getaddrinfo** allows you to specify which types of IP addresses and data that should be retrieved from the DNS server. This is controlled by the *ai\_flags* and the *ai\_family* fields on the hint structure passed to **getaddrinfo**:

<i>ai_flags</i> :	
AI_ALL	Return all IPv4 and IPv6 addresses found. IPv4 addresses will be represented as IPv4-mapped IPv6 addresses.
AI_V4MAPPED	Only return IPv4 addresses if no native IPv6 addresses are found. IPv4 addresses will be represented as IPv4-mapped IPv6 addresses.
AI_ADDRCONFIG	Return IPv4 addresses only if IPv4 is configured on this device; return IPv6 only if IPv6 is configured on this device.
AI_CANONNAME	Requests the canonical name of the given host name. When set, the canonical name is returned in the <i>ai_canonname</i> field of the first returned <i>addrinfo</i> structure.
AI_NUMERICHOST	Don't use name resolution. This flag can be set when passing the ASCII representation of an address to <b>getaddrinfo</b> , which causes it to behave like <b>inet_pton</b> .

These flags allow quite a bit of versatility, but here are some common settings for *ai\_flags* and *ai\_family*:

Only IPv4 addresses:

```

ai_flags = 0;
ai_family = AF_INET;

```

Only IPv6 addresses:

```

ai_flags = 0;
ai_family = AF_INET6;

```

All addresses (IPv4 and IPv6). IPv4 addresses will be represented as IPv4-mapped IPv6 addresses:

```

ai_flags = AI_ALL;
ai_family = AF_INET6;

```

Only return IPv4 addresses if no IPv6 addresses are available. IPv4 addresses will be represented as IPv4-mapped IPv6 addresses:

```

ai_flags = AI_V4MAPPED;
ai_flags = AF_INET6;

```

**Non-blocking mode**

Like all of the other DNS functions, **getaddrinfo** can be used in non-blocking mode by specifying `TM_BLOCKING_OFF` when `tfDnsInit` is called. In non-blocking mode, **getaddrinfo** may return `TM_EWOULDBLOCK`. This indicates that the operation is in progress. Continue to call **getaddrinfo** until a value other than `TM_EWOULDBLOCK` is returned, which indicates that the operation is complete.

For example code using **getaddrinfo**, please see the `examples/txdns.c` file on the distribution CD.

**Parameters**

Parameter	Description
<i>nodeName</i>	String containing nodename to translate.
<i>serviceName</i>	Not currently used.
<i>hintsPtr</i>	Pointer to structure indicating which types of addresses the caller is interested in.
<i>resPtrPtr</i>	Set to a pointer to one or more <i>addrinfo</i> structures containing address information about the specified nodename.

**Returns**

Value	Meaning
<code>TM_ENOERROR</code>	Address information successfully retrieved.
<code>TM_EOPNOTSUPP</code>	Specifying a service name is not supported.
<code>TM_ETIMEDOUT</code>	DNS request failed – requests to DNS server timed out.
<code>TM_EHOSTUNREACH</code>	No route to DNS server.
<code>TM_EWOULDBLOCK</code>	Indicates that an operation is currently in progress and has not yet completed. Only returned in non-blocking mode. (see above)
<code>EAI_NONAME</code>	<code>AI_NUMERICHOST</code> was specified but <i>nodeName</i> was not a valid numeric address for the given family type.
<code>EAI_BADFLAGS</code>	Invalid flags specified in hint structure.
<code>EAI_FAMILY</code>	Invalid protocol family (not <code>PF_INET</code> , <code>PF_INET6</code> or <code>PF_UNSPEC</code> ) in hint structure.
<code>EAI_NODATA</code>	No addresses (of the given address family) associated with the nodename.
<code>EAI_MEMORY</code>	Not enough memory to allocate <i>addrinfo</i> structure(s) to be returned.
<code>TM_DNS_EFORMAT</code>	Error from DNS server: Format error. The name server was unable to interpret the query.
<code>TM_DNS_ESERVER</code>	Error from DNS server: Server failure. The name server was unable to process this query due to a problem with the name server.
<code>TM_DNS_ENAME_ERROR</code>	Error from DNS server: Name error. Meaningful only for responses from an authoritative name server, this code signifies that the domain name referenced in the query does not exist.
<code>TM_DNS_ENOT_IMPLM</code>	Error from DNS server: Not implemented. The name server does not support the requested kind of query.
<code>TM_DNS_EREFUSED</code>	Error from DNS server: Refused. The name server refuses to perform the specified operation for policy reasons. For example, a name server may not wish to provide the information to the particular requester, or a name server may not wish to perform a particular operation (e.g., zone transfer) for particular data.
<code>TM_DNS_EANSWER</code>	No answer received from name server (i.e., response packet received, but it did not contain the answer to our query).

### 4.1.7 getnameinfo

```

int          getnameinfo
(
const struct sockaddr *  addressPtr,
int                    addressLength,
char *                 hostname,
int                    hostnameLength,
char *                 serviceName,
int                    serviceNameLength,
int                    flags
);

```

#### Function Description

Translates an address into its corresponding nodename (hostname). Similar to **gethostbyaddr**. Retrieving a service name or port number is not supported in our implementation of **getnameinfo**, so `serviceName` must be set to `NULL`.

#### Non-blocking mode

Like all of the other DNS functions, **getnameinfo** can be used in non-blocking mode by specifying `TM_BLOCKING_OFF` when `tfDnsInit` is called. In non-blocking mode, **getnameinfo** may return `TM_EWOULDBLOCK`. This indicates that the operation is in progress. Continue to call **getnameinfo** until a value other than `TM_EWOULDBLOCK` is returned, which indicates that the operation is complete.

For example code using **getnameinfo**, please see the `examples/txdns.c` file on the distribution CD.

#### Parameters

Parameter	Description
<i>AddressPtr</i>	Pointer to address to translate.
<i>AddressLength</i>	Length of address structure.
<i>Hostname</i>	Buffer to copy nodename into on completion.
<i>hostnameLength</i>	Size of hostname buffer.
<i>serviceName</i>	Not used.
<i>serviceNameLength</i>	Not used.
<i>flags</i>	<code>NI_NUMERICHOST</code>

#### Returns

Value	Meaning
<code>TM_ENOERROR</code>	Hostname info retrieved successfully.
<code>TM_ETIMEDOUT</code>	DNS request failed – requests to DNS server timed out.
<code>TM_EHOSTUNREACH</code>	No route to DNS server.
<code>TM_EWOULDBLOCK</code>	Indicates that an operation is currently in progress and has not yet completed. Only returned in non-blocking mode. (see above)
<code>EAI_NONAME</code>	<code>NI_NUMERICHOST</code> flag was set, but couldn't convert binary address into string.
<code>EAI_FAMILY</code>	Unrecognized or unsupported address family.
<code>EAI_MEMORY</code>	Memory allocation failure
<code>EAI_FAIL</code>	Non-recoverable error occurred.
<code>EAI_NONAME</code>	No entry found for this address.
<code>TM_DNS_EFORMAT</code>	Error from DNS server: Format error. The name server was unable to interpret the query.
<code>TM_DNS_ESERVER</code>	Error from DNS server: Server failure. The name server was unable to process this query due to a problem with the name server.
<code>TM_DNS_ENAME_ERROR</code>	Error from DNS server: Name error. Meaningful only for responses from an authoritative name server, this code signifies that the domain name referenced in the query does not exist.

TM_DNS_ENOT_IMPLM	Error from DNS server: Not implemented. The name server does not support the requested kind of query.
TM_DNS_EREFSUED	Error from DNS server: Refused. The name server refuses to perform the specified operation for policy reasons. For example, a name server may not wish to provide the information to the particular requester, or a name server may not wish to perform a particular operation (e.g., zone transfer) for particular data.
TM_DNS_EANSWER	No answer received from name server (i.e., response packet received, but it did not contain the answer to our query).

#### 4.1.8 getpeername

This BSD socket API is updated to add support for the IPv6 address type *sockaddr\_in6*. The function prototype is unchanged.

When applications use PF\_INET6 sockets to accept TCP connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system returns the peer's address to the application in the **accept**, **recvfrom**, or **getpeername** call using a *sockaddr\_in6* structure with the peer's IPv4 address encoded as an IPv4-mapped IPv6 address.

([IPV6REQ].R2.18:10, [RFC2553].R3.3:10)

#### 4.1.9 getsockname

This BSD socket API is updated to add support for the IPv6 address type *sockaddr\_in6*. The function prototype is unchanged. ([IPV6REQ].R2.18:10, [RFC2553].R3.3:10)

From the POSIX specification:

“The `getsockname( )` function shall retrieve the locally-bound name of the specified socket, store this address in the *sockaddr* structure pointed to by the address argument, and store the length of this address in the object pointed to by the *address\_len* argument. If the actual length of the address is greater than the length of the supplied *sockaddr* structure, the stored address shall be truncated. If the socket has not been bound to a local name, the value stored in the object pointed to by address is unspecified.”

#### 4.1.10 getsockopt

*This function supports both IPv4 and IPv6*

##### **IPPROTO\_IPV6**

##### **protocolLevel options**

IPV6\_UNICAST\_HOPS

IPV6\_MULTICAST\_IF

IPV6\_MULTICAST\_HOPS

##### **Description**

Refer to [RFC2553].

Refer to [RFC2553]. Get the interface index of the outgoing interface for multicast datagrams sent on this socket. An interface index of 0 indicates there is no outgoing interface for multicast datagrams sent on this socket.

Refer to [RFC2553].

#### **New Return Codes:**

##### **Value**

TM\_EOPNOTSUPP

##### **Meaning**

The specified option (i.e. IPV6\_JOIN\_GROUP, IPV6\_LEAVE\_GROUP) cannot be used with **getsockopt**.

### 4.1.11 if\_indextoname

```
char *          if_indextoname
(
unsigned int    ifindex,
char *         ifname
);
```

#### Function Description

*This function supports both IPv4 and IPv6.*

Refer to [RFC2553]. Maps an interface index into its corresponding interface name

([IPV6REQ].R2.18:10)

#### Parameters

Parameter	Description
<i>ifindex</i>	Interface index
<i>ifname</i>	Interface name. The ifname argument must point to a buffer of at least IF_NAMESIZE bytes into which the interface name corresponding to the specified index is returned.

#### Returns

Value	Meaning
NULL	Failed. Call <b>tfGetSocketError</b> for the specific error code.
!=NULL	Pointer to interface name

If **if\_indextoname** fails, the error code can be retrieved with **tfGetSocketError** which will return one of the following:

#### Error Codes

Value	Meaning
TM_ENXIO	There is no interface corresponding to the specified index.
TM_EINVAL	Bad parameter value.

### 4.1.12 if\_nametoindex

```
unsigned int    if_nametoindex
(
const char *   ifname
);
```

#### Function Description

*This function supports both IPv4 and IPv6.*

Refer to [RFC2553]. Maps an interface name into its corresponding interface index. ([IPV6REQ].R2.18:10)

#### Parameters

Parameter	Description
<i>ifname</i>	Interface name

#### Returns

Value	Meaning
0	Failed. Call <b>tfGetSocketError</b> for the specific error code.
>0	Interface index

If **if\_nametoindex** fails, the error code can be retrieved with **tfGetSocketError** which will return one of the following:

#### Error Codes

Value	Meaning
TM_ENXIO	The specified interface name does not exist.
TM_EINVAL	Bad parameter value.

### 4.1.13 inet\_ntop

```

const char *    inet_ntop
(
int             af,
const void *    src,
char *          dst,
int             size
);

```

#### Function Description

*This function supports both IPv4 and IPv6.*

Refer to [RFC2553]. Converts an IP address from the binary format to the standard text presentation format.

([IPV6REQ].R2.18:10)

#### Parameters

Parameter	Description
<i>af</i>	address family of the IP address specified by <i>src</i> . This is either AF_INET (IPv4) or AF_INET6 (IPv6).
<i>src</i>	Points to the IP address in binary format
<i>dst</i>	Points to a buffer where the function will store the resulting text string
<i>size</i>	Size (in bytes) of the buffer pointed to by <i>dst</i> . For IPv6 addresses, the buffer must be at least 46-bytes (i.e. INET6_ADDRSTRLEN). For IPv4 addresses, the buffer must be at least 16-bytes (i.e. INET_ADDRSTRLEN).

#### Returns

Value	Meaning
NULL	Failed. Call <b>tfGetSocketError</b> for the specific error code.
!=NULL	Pointer to the converted IP address in standard text presentation format.

If **inet\_ntop** fails, the error code can be retrieved with **tfGetSocketError** which will return one of the following:

#### Error Codes

Value	Meaning
TM_EAFNOSUPPORT	<i>af</i> was set to an invalid value for address family; valid values are AF_INET and AF_INET6.
TM_ENOSPC	The size of the result buffer (i.e. <i>size</i> ) is inadequate.

#### 4.1.14 inet\_pton

```

int          inet_pton
(
int          af,
const char * src,
void *      dst
);

```

##### Function Description

*This function supports both IPv4 and IPv6.*

Refer to [RFC2553]. Converts an IP address from the standard text presentation format to the binary format.

([IPV6REQ].R2.18:10)

##### Parameters

Parameter	Description
<i>af</i>	address family of the IP address specified by <i>src</i> . This is either AF_INET (IPv4) or AF_INET6 (IPv6).
<i>src</i>	IP address to convert as null-terminated ASCII text. The format of the address must match the address family.
<i>dst</i>	Pointer to a buffer where the binary format of the converted IP address will be stored by this function. The memory that this points to must be allocated by the caller, i.e. a local variable in the caller's address space, and must match the size requirements of the address family (i.e. 4 bytes for AF_INET, 16 bytes for AF_INET6).

##### Returns

Value	Meaning
1	Success
0	Failed: the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string
-1	Failed. Call <b>tfGetSocketError</b> for the specific error code.

If **inet\_pton** fails with a return code of -1, the error code can be retrieved with **tfGetSocketError** which will return one of the following:

##### Error Codes

Value	Meaning
TM_EAFNOSUPPORT	<i>af</i> was set to an invalid value for address family; valid values are AF_INET and AF_INET6.

**4.1.14.1 inet\_pton Example**

```
struct sockaddr_storage    ipv4Addr;
struct sockaddr_storage    ipv6Addr;
struct sockaddr_storage    ipv6Addr2;
struct sockaddr_in6        *ipv6AddrPtr;
ttUserInterface            interfaceHandle;
int                        errorCode;
...
interfaceHandle = tfAddInterface(...);
...
/* do necessary initialization of sockaddr_storage structure */
tfMemSet(&ipv4Addr, 0, sizeof(struct sockaddr_storage));
ipv4Addr.ss_family = AF_INET; /* IPv4 address */
ipv4Addr.ss_len = sizeof(struct sockaddr_storage);

tfMemSet(&ipv6Addr, 0, sizeof(struct sockaddr_storage));
ipv6Addr.ss_family = AF_INET6; /* IPv6 address */
ipv6Addr.ss_len = sizeof(struct sockaddr_storage);

/* setup to use the IPv4 address 192.168.100.1 with the BSD socket APIs */
errorCode = inet_pton(
    AF_INET,
    "192.168.100.1",
    (void*) &ipv4Addr.addr.ipv4.sin_addr);

/* setup to use the link-local scope IPv6 address
 * fe80:0000:0000:0000:02ea:c0ff:fef0:0860 with the BSD socket APIs
 */
errorCode = inet_pton(
    AF_INET6,
    "fe80:0000:0000:0000:02ea:c0ff:fef0:0860",
    (void*) &ipv6Addr.addr.ipv6.sin6_addr);

/* do the same thing is a way which is completely BSD-compliant (i.e.
 * without the Treck extensions to sockaddr_storage), and
 * abbreviate the IPv6 address to a shorter form
 */
tfMemSet(&ipv6Addr2, 0, sizeof(struct sockaddr_storage));
ipv6AddrPtr = (struct sockaddr_in6 *) &ipv6Addr2;
ipv6AddrPtr->sin6_family = AF_INET6; /* IPv6 address */
ipv6AddrPtr->sin6_len = sizeof(struct sockaddr_in6);
errorCode = inet_pton(
    AF_INET6,
    "fe80::2ea:c0ff:fef0:860",
```

```
(void*) &ipv6AddrPtr->sin6_addr);

/* WARNING: this isn't good enough, because the address is link-local
 * scope. We must also set the sin6_scope_id field to identify the
 * interface that it is scoped to.
 */
errorCode = tf6SockaddrSetScopeId(
    interfaceHandle, &ipv6Addr2);
```

#### 4.1.15 `recvfrom`

This BSD socket API is updated to add support for the IPv6 address type `sockaddr_in6`. The function prototype is unchanged.

When applications use `PF_INET6` sockets to accept TCP connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system returns the peer's address to the application in the `accept`, `recvfrom`, or `getpeername` call using a `sockaddr_in6` structure with the peer's IPv4 address encoded as an IPv4-mapped IPv6 address. ([IPV6REQ].R2.18:10, [RFC2553].R3.3:10)

#### 4.1.16 `sendto`

This BSD socket API is updated to add support for the IPv6 address type `sockaddr_in6`. The function prototype is unchanged.

Applications may use `PF_INET6` sockets to open TCP connections to IPv4 nodes, or send UDP packets to IPv4 nodes, by simply encoding the destination's IPv4 address as an IPv4-mapped IPv6 address, and passing that address, within a `sockaddr_in6` structure, in the `connect` or `sendto` call. ([IPV6REQ].R2.18:10)

#### 4.1.17 `setsockopt`

##### *IPPROTO\_IPV6*

##### *protocolLevel options*

`IPV6_UNICAST_HOPS`

`IPV6_MULTICAST_IF`

`IPV6_MULTICAST_HOPS`

`IPV6_JOIN_GROUP`

`IPV6_LEAVE_GROUP`

##### **Description**

Refer to [RFC2553]. Stored in `ttSocketEntry.soc6HopLimit`.

Refer to [RFC2553]. Specify the interface index of the outgoing interface for multicast datagrams sent on this socket. An interface index of 0 indicates that we want to reset a previously set outgoing interface for multicast datagrams sent on this socket. Pointer to the interface/device is stored in `ttSocketEntry.soc6McastDevPtr`.

Refer to [RFC2553]. Stored in `ttSocketEntry.soc6McastHopLimit`. The default value is 1 (i.e. `TM_6_IP_DEF_MULTICAST_HOPS`).

Refer to [RFC2553]. Join IPv6 multicast group. (see struct `ipv6_mreq` data type below)

Refer to [RFC2553]. Leave IPv6 multicast group. (see struct `ipv6_mreq` data type below)

```

struct          ipv6_mreq
{
    struct in6_addr  ipv6mr_multiaddr;
    unsigned int    ipv6mr_interface;
};

```

#### 4.1.18 `socket`

This BSD socket API is updated to add support for protocol family `PF_INET6` (IPv6). Code must be added to populate the new field `ttSocketEntry.socProtocolFamily`. IPv6 must support the socket type `SOCK_RAW` since the Ping API requires it. The function prototype is unchanged. ([IPV6REQ].R2.18:10).

If you want to use ICMPv6 with an IPv6 raw socket (i.e. socket type is `SOCK_RAW`, socket family is `AF_INET6`), then the socket protocol must be `IPPROTO_ICMPV6`. Note that ICMP is not an IPv6 protocol, and cannot be used with IPv6.

#### **New Return Codes:**

<b>Value</b>	<b>Meaning</b>
<code>TM_EAFNOSUPPORT</code>	The specified address family is not supported.

## 4.2 Socket Extension Calls

The following APIs are fully described in **Chapter 5** of the *Treck TCP/IP User's Manual*.

The manner in which these APIs are used has remained the same, but additional functionality has been added for IPv6 support.

### 4.2.1 tf6AddressToIpv4Compat

```
int                tf6AddressToIpv4Compat
(
  ttUserIpAddress  inIpv4Addr,
  struct sockaddr_storage * outIpv6AddrPtr
);
```

#### Function Description

*This function only supports IPv6.*

This function converts an IPv4 address into an IPv4-compatible IPv6 address, typically for use with automatic tunneling.

#### Parameters

Parameter	Description
<i>inIpv4Addr</i>	IPv4 address to convert.
<i>outIpv6AddrPtr</i>	Pointer to converted IPv4-compatible IPv6 address. The memory that this points to must be allocated by the caller, i.e. a local variable in the caller's address space.

#### Returns

Value	Meaning
0	Success.
TM_EINVAL	Invalid IPv4 address specified. IPv4 address must be global unicast address.
TM_EINVAL	<i>outIpAddrPtr</i> was NULL.

### 4.2.2 `tf6AddressToIpv4Mapped`

```
int                                     tf6AddressToIpv4Mapped  
(  
  ttUserIpAddress                     inIpv4Addr,  
  struct sockaddr_storage * outIpv6AddrPtr  
);
```

#### Function Description

*This function only supports IPv6.*

This function converts an IPv4 address into an IPv4-mapped IPv6 address (which is an IPv4-only address represented in 128-bit format).

#### Parameters

Parameter	Description
<i>inIpv4Addr</i>	IPv4 address to convert.
<i>outIpv6AddrPtr</i>	Pointer to converted IPv4-mapped IPv6 address. The memory that this points to must be allocated by the caller, i.e. a local variable in the caller's address space.

#### Returns

Value	Meaning
0	Success.
TM_EINVAL	<i>outIpAddrPtr</i> was NULL.

### 4.2.3 `tf6GetLocalIpAddress`

```

int                                     tf6GetLocalIpAddress
(
ttUserInterface                         interfaceHandle,
const struct sockaddr_storage *         destIpAddressPtr,
tt6LocalIpAddressCursorPtr            addrCursorPtr,
int                                     initCursorFlag,
struct sockaddr_storage *              localIpAddressPtr
);

```

#### Function Description

*This function only supports IPv6.*

This function is called to retrieve a local IP address appropriate for use/communication with the specified destination IP address from the list of multi-homed IPv6 addresses associated with the specified interface. There may be more than one matching local IP address in that list for the specified interface and destination IP address, in which case this function may be called multiple times to retrieve each appropriate local IP address one at a time, setting *initCursorFlag* appropriately as described below. Alternatively, if the caller only wants to retrieve the first/best match, *addrCursorPtr* may be set to NULL to indicate that no iteration is desired.

The default mode of operation is to return preferred IPv6 addresses first, and next to return deprecated IPv6 addresses if there are any. You can disable the use of deprecated addresses by calling **tf6SetTreckOptions** to enable the option `TM_6_OPTION_IP_DEPRECATE_ADDR`, in which case **tf6GetLocalIpAddress** does not return deprecated IPv6 addresses. Note that after this function is called, **bind** must still be called to associate the local IP address with a socket so that it is used in communications on that socket.

When the destination IPv6 address is a global scope unicast IPv6 address, this function may be called to search all interfaces for the best matching local IPv6 address by setting *interfaceHandle* to NULL.

This API has visibility to all IPv6 addresses configured on the interface, including those auto-configured by IPv6 stateless address auto-configuration.

([RFC2462].R4.1:10, [RFC2893].R5.5:10, [RFC2893].R5.5:20, [RFC2893].R5.5:30, [RFC2462].R5.5.4:50, [RFC2462].R5.5.4:30, [RFC3056].R2.1:10, [RFC3056].R2.1:20)

#### Parameters

##### Parameter

*interfaceHandle*

##### Description

Interface handle of the outgoing interface for which we want to select an appropriate local IPv6 address. When the destination IPv6 address is a global scope unicast IPv6 address, *interfaceHandle* may be set to NULL, in which case the outgoing interface is determined by doing a longest prefix match on the destination IPv6 address, and then the global scope IPv6 addresses configured on that interface are returned in longest prefix match order.

*destIpAddressPtr*

Pointer to the destination IPv6 address with which we want to communicate.

*addrCursorPtr*

Pointer to a cursor used to iterate through the list of multi-homed IPv6 addresses associated with the specified interface. *addrCursorPtr* may be set to NULL to indicate that no iteration is desired, otherwise the memory that this points to must be allocated by the caller, i.e. a local variable of type `tt6LocalIpAddressCursor` in the caller's address space. This function uses the cursor to keep track of where to restart the search in the multi-homed IPv6 address list each time it is called to find the next appropriate local IP address. Therefore, the caller must not change the cursor value, since this function updates it each time it is called to keep track of the state of the iteration.

*initCursorFlag*

Must be set to 1 (i.e. non-zero) the first time this function is called for a specific combination of interface and destination IP address, otherwise set to 0. When *initCursorFlag* is set to 1, this function initializes the cursor (pointed to by *addrCursorPtr*) to point to the beginning of the list of multi-homed IPv6 addresses associated with the specified interface, and therefore starts the search for an appropriate local IP address at the beginning of this list. To retrieve additional appropriate IP addresses, *initCursorFlag* must be set to 0 on subsequent calls to this function so that the cursor can properly iterate through the list.

*localIpAddrPtr*

When this function returns 0 (i.e. success), *localIpAddrPtr* points to a local IP address appropriate for use/communication with the specified destination IP address. The memory that this points to must be allocated by the caller, i.e. a local variable in the caller's address space. If there is more than one appropriate local/source IP address, they may be retrieved individually by calling this function multiple times, specifying the same interface and destination IP address each time, and each time specifying the correct value for *initCursorFlag* as described above.

**Returns****Value**

0

TM\_EINVAL

TM\_ENETDOWN

TM\_ENOENT

TM\_EAFNOSUPPORT

**Meaning**

Success.

One of the parameters was invalid. Either an invalid interface handle was specified, or the cursor (pointed to by *addrCursorPtr*) is set to an invalid value and *initCursorFlag* is not 1, or one of the pointers passed in as parameters (i.e. *destIpAddrPtr*, *addrCursorPtr*, *selectedLocalIpAddrPtr*) was NULL, or the specified destination IP address (pointed to by *destIpAddrPtr*) was an invalid IPv6 address.

Interface is not configured.

There was no appropriate local IP address found; iteration of the multi-homed IPv6 address list is complete.

The specified destination IP address (pointed to by *destIpAddrPtr*) was not an IPv6 address, i.e. *destIpAddrPtr->addrFamily* != AF\_INET6

#### 4.2.4 `tf6SockaddrSetScopeId`

```
int                tf6SockaddrSetScopeId
(
  ttUserInterface  interfaceHandle,
  struct sockaddr_storage * ipAddrPtr
);
```

#### Function Description

*This function only supports IPv6.*

In some cases you may need to set the scope ID of a local scope unicast IPv6 address to associate it with the specific interface that you want to use it with. For example, if you want to send packets to a link-local or site-local unicast IPv6 destination address, you need to associate the local scope destination address with the specific outgoing interface you want to send those packets on, and this is done via the scope ID. In that case, you can call `tf6SockaddrSetScopeId` to set the scope ID of the address. This function may be called for other IPv6 addresses besides local scope unicast IPv6 addresses, in which case it will return a scope ID of 0.

#### Parameters

Parameter	Description
<i>interfaceHandle</i>	Interface handle of the interface to associate with the local scope IPv6 address.
<i>ipAddrPtr</i>	Pointer to the IPv6 address that you want to set the scope ID for. The memory that this points to must be allocated by the caller, i.e. a local variable in the caller's address space. On successful return, the <code>sockaddr_in6.sin6_scope_id</code> field of this address is set to the scope ID.

#### Returns

Value	Meaning
0	Success.
TM_EINVAL	One of the parameters was invalid.
TM_EAFNOSUPPORT	The specified IP address (pointed to by <i>inIpAddrPtr</i> ) was not an IPv6 address, i.e. <code>inIpAddrPtr-&gt;addrFamily != AF_INET6</code>

#### 4.2.4.1 *tf6SockaddrSetScopeId Example*

```
    struct sockaddr_storage  peerIpNgAddr;
    ttUserInterface         interfaceHandle;
    int                     pingSd;

...
/* setup to ping the IPv6 peer address 3FFE::12:99:240:219 */
    errorCode = inet_pton(
        AF_INET6, "3FFE::12:99:240:219",
        &(peerIpNgAddr.addr.ipv6.sin6_addr));

/* if the address is local scope, we must scope it to a specific
 * interface, otherwise the Treck stack will not know how to route to it.
 * Call tf6SockaddrSetScopeId to do this. Note: tf6SockaddrSetScopeId
 * works correctly regardless of the scope of the IPv6 address.
 */
    errorCode = tf6SockaddrSetScopeId(
        interfaceHandle, &peerIpNgAddr);

/* start the ping */
    pingSd = tfNgPingOpenStart(
        &peerIpNgAddr,
        1000, /* 1 second between retransmissions */
        100, /* 100 bytes of data */
        (ttPingCBFuncPtr) 0);

...

```

## 4.3 Treck Initialization Functions

### 4.3.1 `tfInItTreckOptions`

This function supports both IPv4 and IPv6. Refer to `tfSetTreckOptions` below.

### 4.3.2 `tfSetTreckOptions`

This function supports both IPv4 and IPv6.

---

Please See `tfSetTreckOptions` in Chapter 5 'Programmer's Reference of the Treck TCP/IP User's Manual'. The following are additional configuration options for IPv6 Support.

---

Option Name	Meaning
<code>TM_6_OPTION_ICMP_TX_ERR_LIMIT</code>	Specifies the number of milliseconds between sending ICMP error messages. Default is 500, which means that the maximum frequency for sending ICMP error messages is 2 per second. ([RFC2463].R2.4:110)
<code>TM_6_OPTION_IP_DEPRECATE_ADDR</code>	Enable/disable prevention of any new IPv6 communication from using a deprecated local IPv6 address. Disabled by default. ([RFC2462].R5.5.4:50, [RFC2462].R5.5.4:60, [RFC2462].R5.5.4:70)
<code>TM_6_OPTION_IP_FORWARDING</code>	A boolean used to enable IPv6 forwarding; indicates whether the IPv6 node is operating as a host or as a router. Disabled by default. Note that router functionality is not implemented in phase 1 IPv6.
<code>TM_6_OPTION_IP_FRAGMENT</code>	A boolean used to enable IPv6 fragmentation. Enabled by default.
<code>TM_6_OPTION_ND_MAX_UNICAST_RETRY</code>	The maximum number of Neighbor Solicitations to send while probing a neighbor for reachability detection. <b>DEFAULT: 3.</b>
<code>TM_6_OPTION_ND_MAX_MCAST_RETRY</code>	The maximum number of Neighbor Solicitations to send while doing address resolution. <b>DEFAULT: 3.</b>
<code>TM_6_OPTION_IP_FRAG_TTL</code>	IPv6 fragment reassembly timeout in seconds. <b>DEFAULT: 64.</b>
<code>TM_6_OPTION_IP_FRAG_MAX_Q_SIZE</code>	The maximum number of IPv6 datagrams waiting to be reassembled. If any fragment of a new datagram arrives when the maximum number of IP datagrams waiting to be reassembled has been reached, it is dropped. <b>DEFAULT: 5.</b>
<code>TM_6_OPTION_IP_FRAG_ENTRY_MAX_SIZE</code>	The maximum size of an IPv6 datagram waiting to be reassembled. Beyond that size, we drop the entire IP datagram. <b>DEFAULT: 8200 bytes.</b>
<code>TM_6_OPTION_PATH_MTU_TTL</code>	The time (in seconds) before an IPv6 Path MTU estimate is increased, in order to discovery a larger Path MTU value. According to RFC-1981, this value should never be set less than 5 minutes. To set this timeout to "infinity" use a value of <code>TM_RTE_INF</code> . This will effectively disable the periodic increasing of PMTU entries. ([RFC1981].R4:30, [RFC1981].R.5.3:20) <b>Default: 600 seconds</b>

## 4.4 Device / Interface API

### 4.4.1 tf4NetmaskToPrefixLen

```
int          tf4NetmaskToPrefixLen
(
  ttUserIpAddress netmask
);
```

#### Function Description

*This function only supports IPv4.*

Convert an IPv4 netmask into a prefix length. This API is provided to facilitate porting of existing code to the tfNg APIs.

#### Parameters

Parameter	Description
<i>netMask</i>	The IPv4 netmask to get the prefix length for.

#### Returns

Returns the 8-bit prefix length, which is the number of high-order bits in the netmask that are set to one.

#### 4.4.2 tf6Eui64GetInterfaceId

```
int
ttUserInterface
ttUser8Bit *
);

tf6Eui64GetInterfaceId(
interfaceHandle,
eui64IdPtr
```

##### Function Description

*This function only supports IPv6.*

This function is called to retrieve the value of the 64-bit interface ID previously set on the specified IPv6 interface. Note that the interface ID is set automatically by the Ethernet link-layer when the interface is opened for use with IPv6, otherwise the interface ID must first be manually set by the user calling **tf6Eui64SetInterfaceId**. The interface ID returned by this function is already formatted as **tf6Eui64SetInterfaceId** expects.

##### Parameters

Parameter	Description
<i>interfaceHandle</i>	Interface handle of an interface for which we want to get the interface ID.
<i>eui64IdPtr</i>	Pointer to the 8-byte interface ID in EUI-64 format (i.e. 64-bit IEEE global identifier). The memory that this points to must be allocated by the caller, i.e. a local variable in the caller's address space.

##### Returns

Value	Meaning
0	Success. <i>eui64IdPtr</i> now points to the interface ID.
TM_EINVAL	Invalid parameter value. Either <i>interfaceHandle</i> does not point to a valid interface, or <i>eui64IdPtr</i> is a NULL pointer.
TM_ENOENT	There is no interface ID set on the specified interface. If the interface uses the Ethernet link-layer, then open the interface for use with IPv6, otherwise call <b>tf6Eui64SetInterfaceId</b> to set an interface ID on the interface.

### 4.4.3 `tf6Eui64SetInterfaceId`

```

int                tf6Eui64SetInterfaceId
(
  ttUserInterface  interfaceHandle,
  const ttUser8Bit * eui64IdPtr,
  int              calledFromFlag
);

```

#### Function Description

*This function only supports IPv6.*

This function sets or resets the IPv6 interface ID to the specified interface ID in EUI-64 format (i.e. 64-bit IEEE global identifier). Note that this function inverts the “u” (universal/local) bit of the EUI-64 identifier to convert it into an interface ID. If your EUI-64 identifier has local significance only (i.e. it is not globally unique), then to indicate this you must set bit 6 to “1” where bit 0 is the most significant bit. A universally administered (i.e. globally unique) EUI-64 identifier is indicated by setting bit 6 to “0”. ([RFC2464].R4:20, [RFC2373].R2.4:20)

This function may only be called on an interface that is closed or that has IPv6 disabled. When the Treck stack is running in dual IP layer mode, “IPv6 disabled” means that the interface is opened for IPv4, and therefore the link-layer and device driver are open (i.e. IPv4 and IPv6 share the same link-layer and device driver on the interface). However, there is no link-local IPv6 address configured on the interface so the IPv6 part of the interface cannot be used by the application. Typically, this is the case when you have not yet opened the interface for IPv6 by calling `tfNgOpenInterface` specifying an IPv6 address to configure. When the Treck stack is running in IPv6-only mode, “IPv6 disabled” means that you opened the interface for IPv6, however either the link-local address generated by IPv6 stateless address auto-configuration was detected to be in use by a different node (i.e. Duplicate Address Detection failed), or you intentionally unconfigured the link-local address by calling `tfNgUnConfigInterface`. In any case, there is no link-local IPv6 address configured on the interface, so the IPv6 part of the interface cannot be used by the application.

When the interface has IPv6 disabled, recovery usually entails either setting a different interface ID by calling `tf6Eui64SetInterfaceId` and then attempting to restart the IPv6 part of the interface by calling `tfNgOpenInterface` specifying an IPv6 address of all 0's (i.e. the IPv6 unspecified address), or by manually configuring a link-local IPv6 address on the interface. However, it is possible that you get this failure because your network controller hardware does not support filtering of its own multicasts but instead does loopback of its own multicasts, which would cause Duplicate Address Detection to always fail. If this is the case, then you must specify the IPv6-specific device flag `TM_6_DEV_MCAST_HW_LOOPBACK` when you open the interface. ([RFC2373].R2.1:10)

The IPv6 interface ID is used to generate tentatively unique IPv6 addresses to assign to the interface during stateless address auto-configuration, which occurs when the interface is opened. Part of stateless address auto-configuration is Duplicate Address Detection, which is done to ensure that the auto-configured IPv6 addresses are unique. Note that if an interface ID of 0 is set, stateless address auto-configuration of global scope and site-local scope IPv6 addresses will not be performed when the interface is opened, because otherwise the auto-configured IPv6 addresses would conflict with the Subnet-Router anycast addresses for the same prefixes. In this case, the only auto-configured IPv6 address assigned to the interface by stateless address auto-configuration could be the link-local scope IPv6 address. In other words, 0 is a bad choice for the interface ID. ([RFC2373].R2.6.1:10)

If you are using the null link-layer and you want to use IPv6 on the interface, then you should call `tf6Eui64SetInterfaceId`, typically done in your device driver open function in which case `calledFromFlag` is set to `TM_6_DEV_CALLED_FROM_DRIVER`. Otherwise, if the interface ID is not set before the interface is opened, then IPv6 stateless address auto-configuration will not be performed, which may result in the IPv6 interface not being opened depending on what IPv6 addresses have been manually configured on the interface (i.e. at least one link-local IPv6 address is required). ([RFC2373].R2.1:10)

If your interface link-layer type is Ethernet, then you should not call this function, since the interface ID will be automatically derived from the IEEE 48-bit MAC address you provide via your `driverGetPhysicalAddress` function (refer to the `drvGetPhysAddrFuncPtr` parameter of the function `tfAddInterface`).

**Parameters****Parameter***interfaceHandle**eui64IdPtr**calledFromFlag***Description**

Interface handle of an interface for which we want to set the interface ID  
Pointer to the 8-byte interface ID in EUI-64 format (i.e. 64-bit IEEE global identifier).

Specifies whether this function is being called from the device driver, typically from the device driver open function. It is very important to set this correctly. If you are calling this function from the device driver, set *calledFromFlag* to `TM_6_DEV_CALLED_FROM_DRIVER`, otherwise set it to `TM_6_DEV_CALLED_FROM_APP`.

**Returns****Value**

0

TM\_EINVAL

TM\_EPERM

**Meaning**

Success.

One of the parameters was invalid.

The interface specified by *interfaceHandle* was open for IPv6, and the IPv6 part was not disabled. This function may only be called on an interface that is closed or that has IPv6 disabled.

#### 4.4.4 `tf6InterfaceSetPhysAddr`

```

int                tf6InterfaceSetPhysAddr
(
ttUserInterface   interfaceHandle,
const ttUser8Bit * physAddrPtr,
int               physAddrLen,
int               calledFromFlag
);

```

##### Function Description

*This function only supports IPv6.*

This function sets the link-layer address associated with the interface, which allows the interface to initiate and respond to requests for IPv6 address resolution. If you specify a non-null value for the `drvGetPhyAddrFuncPtr` parameter in the call to **tfAddInterface** and you are using the Ethernet link-layer, then you do not need to call this function, since the link-layer address is obtained by calling your `driverGetPhysicalAddress` function.

This function may only be called on an interface that is closed, and is typically called from your device driver open function, in which case `calledFromFlag` is set to `TM_6_DEV_CALLED_FROM_DRIVER`.

Note that if you use **tf6InterfaceSetPhysAddr** instead of `drvGetPhyAddrFuncPtr` to set the link-layer address, then you should also call **tf6Eui64SetInterfaceId** to set the interface ID. ([RFC2461].R7.2.6:10, [RFC2461].R7.2.6:20)

##### Parameters

Parameter	Description
<code>interfaceHandle</code>	Interface handle of an interface for which we want to set the link-layer address.
<code>physAddrPtr</code>	Pointer to the link-layer address.
<code>physAddrLen</code>	Length in bytes of the link-layer address.
<code>calledFromFlag</code>	Specifies whether this function is being called from the device driver, typically from the device driver open function. It is very important to set this correctly. If you are calling this function from the device driver, set <code>calledFromFlag</code> to <code>TM_6_DEV_CALLED_FROM_DRIVER</code> , otherwise set it to <code>TM_6_DEV_CALLED_FROM_APP</code> .

##### Returns

Value	Meaning
0	Success.
<code>TM_EINVAL</code>	One of the parameters was invalid, or <code>physAddrLen</code> was greater than <code>TM_MAX_PHYS_ADDR</code> .
<code>TM_EPERM</code>	The interface specified by <code>interfaceHandle</code> was open. This function may only be called on an interface that is closed.

#### 4.4.5 tf6InterfaceSetSiteId

```
int          tf6InterfaceSetSiteId
(
ttUserInterface  interfaceHandle,
ttUser32Bit      siteId
);
```

##### Function Description

*This function only supports IPv6.*

This function is used to change the IPv6 site identifier associated with an interface. The site identifier is used as the scope ID for site-local unicast IPv6 addresses associated with the specified interface. This function may only be called on an interface that is closed, i.e. before **tfNgOpenInterface** or **tfNgConfigInterface** is called.

##### Parameters

Parameter	Description
<i>interfaceHandle</i>	Interface handle of an interface for which we want to set the site identifier.
<i>siteId</i>	The site identifier. Must be non-zero.

##### Returns

Value	Meaning
0	Success.
TM_EINVAL	One of the parameters was invalid.
TM_EINVAL	<i>siteId</i> was 0. <i>siteId</i> must be non-zero.
TM_EPERM	The interface specified by <i>interfaceHandle</i> was open. This function may only be called on an interface that is closed.

### 4.4.6 tf6SetMcastInterface

```
int          tf6SetMcastInterface
(
ttUserInterface  interfaceHandle
);
```

#### Function Description

*This function only supports IPv6.*

This function sets the default interface to use to send IPv6 multicast packets. The application can set the IPV6\_MULTICAST\_IF option on the socket to override this default interface on a per-socket basis.

#### Parameters

Parameter	Description
<i>interfaceHandle</i>	Interface handle of the default interface to use to send IPv6 multicast packet.

#### Returns

Value	Meaning
0	Success
TM_EINVAL	Invalid interface handle.

#### 4.4.7 tf6SetRandomInterfaceId

```
int          tf6SetRandomInterfaceId(
ttUserInterface interfaceHandle
);
```

##### Function Description

*This function only supports IPv6.*

This function sets a random interface ID per [RFC3041]. Note that the specified interface must be closed or disabled for use with IPv6 when you call this API. This API can be used to recover from the Duplicate Address Detection failure that could occur when auto-configuring a link-local scope IPv6 address on the interface.

([MIPV6\_19].R7.7:40.10)

##### Parameters

Parameter	Description
<i>interfaceHandle</i>	Interface handle of an interface that we want to set a random interface ID on.

##### Returns

Value	Meaning
0	Success.
TM_EINVAL	You specified an invalid interface handle.
TM_EPERM	The interface specified by <i>interfaceHandle</i> was open for IPv6, and the IPv6 part was not disabled. This function may only be called on an interface that is closed or that has IPv6 disabled.

#### 4.4.8 tfCheckOpenInterface

```
int          tfCheckOpenInterface
(
ttUserInterface  interfaceHandle,
int             protocolFamily
);
```

##### Function Description

This function supports both IPv4 and IPv6.

Check the status of either IPv4 or IPv6 protocol operation on the specified interface.

##### Parameters

Parameter	Description
<i>interfaceHandle</i>	Interface handle of the interface to check the status for.
<i>protocolFamily</i>	The protocol family (i.e. IPv4 or IPv6) to check the status for. Specify PF_INET for IPv4, PF_INET6 for IPv6.

##### Returns

Value	Meaning
0	Interface is open/up for the specified protocol family (i.e. IPv4 or IPv6).
TM_EINPROGRESS	The interface is still being opened for the specified protocol family. If the interface is being opened for use with IPv6, then this error code is returned while Duplicate Address Detection is being performed for the link-local scope IPv6 address.
TM_ENETDOWN	Interface is closed for the specified protocol family.
TM_EPERM	Interface is disabled for the specified protocol family. When <i>protocolFamily</i> is PF_INET6, this indicates that there is no link-local scope IPv6 address configured on the interface, however the interface is not closed, instead the IPv6 part of the interface is disabled and the application cannot use IPv6 on this interface. This could occur if the link-local IPv6 address generated by IPv6 stateless address auto-configuration was determined to already be in use by a different node (i.e. Duplicate Address Detection failed), in which case recovery usually entails either setting a different interface ID, or manually configuring a link-local IPv6 address on the interface, and then attempting to restart the IPv6 part of the interface by calling <b>tfNgOpenInterface</b> specifying an IPv6 address of all 0's (i.e. the IPv6 unspecified address).
TM_EINVAL	Bad parameter value. Either you specified an invalid interface handle, or the protocol family was not PF_INET or PF_INET6, or the protocol family did not match the mode in which you are running the stack (for example, if you run the stack in IPv4-only mode and specify a <i>protocolFamily</i> of PF_INET6, you will get this error).

#### 4.4.8.1 *tfCheckOpenInterface Example*

```
ttUserInterface interfaceHandle;
struct sockaddr_storage ipNgAddr;
...

/* open the interface for use with IPv6:
 * start IPv6 stateless address auto-configuration to get a link-local
 * scope IPv6 address assigned to the interface.
 */
tfMemSet(&ipNgAddr, 0, sizeof(struct sockaddr_storage));
ipNgAddr.ss_family = AF_INET6; /* IPv6 address */
ipNgAddr.ss_len = sizeof(struct sockaddr_storage);
errorCode=tfNgOpenInterface(
    interfaceHandle,
    &ipNgAddr, /* IPv6 address is all 0's, auto-configuration */
    0, /* prefix length does not matter for auto-configuration */
    configFlags,
    0, /* IPv6-specific flags */
    scatteredBufferCount,
    TM_6_DEV_ADDR_NOTIFY_FUNC_NULL_PTR);

/* wait for auto-configuration to complete */
while (tfCheckOpenInterface(interfaceHandle, AF_INET6)
    != TM_ENOERROR)
{
    OSTimeDly(1); /* allow other tasks to run for 10 milliseconds */
}
```

#### 4.4.9 `tfInterfaceSetOptions`

*This function supports both IPv4 and IPv6*

Add the following new options to the existing documentation for `tfInterfaceSetOptions`:

Option Name	Data Type	Description
<code>TM_6_DEV_OPTIONS_DAD_XMITS</code>	unsigned char	<p><i>DupAddrDetectTransmits:</i></p> <p>Controls the number of consecutive Neighbor Solicitation messages sent while performing Duplicate Address Detection on a tentative address. When set to 0, disables Duplicate Address Detection for tentative addresses on the specified interface. Default value is 1. Option can only be set when the device/interface is closed. ([RFC2462].R5.1:10, [RFC2462].R5.1:20, [RFC2462].R5.1:30, [RFC2462].R5.1:40, [RFC2462].R5.1:50)</p>
<code>TM_6_DEV_OPTIONS_NO_DHCP_CONF</code>	unsigned char	<p>By default, when the stack is operating in dual IPv4/IPv6 mode and a global IPv4 address is acquired through BOOTP/DHCP, an IPv4-compatible IPv6 address corresponding to the acquired address is also configured. Setting this option to '1' disables this behavior. <b>Default: 0</b></p>

#### 4.4.10 `tfNgConfigInterface`

```

int                                     tfNgConfigInterface
(
ttUserInterface                       interfaceHandle,
const struct sockaddr_storage * ipAddrPtr,
int                                     prefixLen,
int                                     flags,
int                                     ipv6Flags,
int                                     buffersPerFrameCount,
tt6DevAddrNotifyFuncPtr               dev6AddrNotifyFuncPtr,
unsigned int                            multiHomeIndex
);

```

##### Function Description

Next Generation API replaces `tfConfigInterface`. If it has not already been started, this API starts the IPv4 part of an interface if the specified address is an IPv4 address (or an IPv4-mapped IPv6 address), starts the IPv6 part of an interface if the specified address is an IPv6 address, and starts both IPv4 and IPv6 if the specified address is an IPv4-compatible IPv6 address.

Starting the IPv6 part of an interface causes IPv6 stateless address auto-configuration to be invoked, if an interface ID has been set on the interface, to dynamically assign IPv6 addresses to the interface. If an IPv6 address is specified, it may be all 0's (i.e. the IPv6 unspecified address), in which case no IPv6 address is manually configured on the interface. However, the IPv6 part of the interface is started to attempt IPv6 stateless address auto-configuration. Note that in this case, the device flag `TM_DEV_IP_USER_BOOT` should not be specified since IPv6 does not have any equivalent of the API `tfFinishOpenInterface`.

When you are running the Treck stack in dual IP layer mode and the specified address is an IPv4-compatible IPv6 address, it is configured on the interface at the same multi-home index in both the IPv4 and IPv6 address lists to support automatic tunneling. For example, if you call this API to configure an IPv4-compatible IPv6 address at multi-home index 3, this API checks both IPv6 and IPv4 address lists to make sure that the slot indexed by multi-home index 3 is available in both, then configures the IPv4-compatible IPv6 address in the IPv6 address list at multi-home index 3, and the embedded IPv4 address (i.e. the low-order 32 bits of the IPv6 address) in the IPv4 address list at multi-home index 3. This is done to support automatic tunneling since automatic tunneling requires us to implement the IPv4 endpoint of the tunnel on the interface and the tunnel endpoint is the embedded IPv4 address. If the slot in the IPv4 address list indexed by the specified multi-home index is already configured with a different IPv4 address, this API returns `TM_ENOSPC` to indicate that configuration of the IPv4-compatible IPv6 address failed because the specified multi-home index was not available to configure. When configuring an IPv4-compatible IPv6 address, the specified prefix length must be the IPv6 prefix length, rather than the IPv4 prefix length. That is, the configured prefix length for an IPv4-compatible IPv6 address must be greater than 96.

When the stack is operating in dual IPv4/IPv6 mode, in order to support automatic tunneling, if a global IPv4 address is acquired via BOOTP/DHCP, an IPv4-compatible IPv6 address corresponding to the acquired address is also configured at the same IPv6 multihome index. For instance, if the first IPv4 multihome entry is configured via DHCP, the stack will automatically attempt to configure a corresponding IPv4-compatible IPv6 address at the first IPv6 multihome entry. However, if another IPv6 address is already configured at the first multihome entry, the IPv4-compatible IPv6 address cannot be added. Please note that this behavior may be disabled with the `TM_6_DEV_OPTIONS_NO_DHCP_CONF` option through `tfInterfaceSetOptions`.

Since this API is used to manually configure addresses, *multiHomeIndex* may not be greater than or equal to `TM_MAX_IPS_PER_IF`.

Note that the macro `TM_MAX_IPS_PER_IF` is not publicly accessible unless you explicitly `#define` it in your `trsystem.h` file.

If *dev6AddrNotifyFuncPtr* is non-null, then the user wants to be notified of IPv6-specific address configuration events via a user-defined notify function. IPv6 is different from IPv4 with regards to address configuration, because every IPv6 address configured on the interface must first go through Duplicate Address Detection to confirm that no other node is using that tentative address before the address may be configured on the interface ([RFC2462].R5.4:10). Also, IPv6

addresses which are auto-configured using stateless address auto-configuration have preferred and valid lifetimes, and when they become invalid, they are unconfigured from the interface. The address configuration events you can be notified of consist of Duplicate Address Detection, as well as IPv6 address state transitions from preferred to deprecated, and to invalid:

Address Configuration Event	Description
TM_6_DEV_ADDR_CONFIG_STARTED	Configuration of the IPv6 address has been started. Duplicate Address Detection is being performed on the tentative address.
TM_6_DEV_ADDR_CONFIG_FAILED	Duplicate Address Detection failed. A different node is already using the tentative address we tried to configure, therefore configuration failed.
TM_6_DEV_ADDR_CONFIG_COMPLETE	Duplicate Address Detection succeeded. Configuration of the IPv6 address on the interface completed successfully.
TM_6_DEV_ADDR_DUP_DETECTED	The Treck stack detected that a different node is using one of your IPv6 addresses that was successfully configured (i.e. for which the TM_6_DEV_ADDR_CONFIG_COMPLETE event has occurred). It is up to the user to specify what recovery, if any, is performed when this event occurs (see below). ([RFC2462].R5.4.4:10)
TM_6_DEV_ADDR_DEPRECATED	An auto-configured IPv6 address has transitioned state from preferred to deprecated.
TM_6_DEV_ADDR_INVALIDATED	An auto-configured IPv6 address has transitioned state to invalid, and has been unconfigured from the interface.

After an IPv6 address has been configured on the interface there is still the possibility that another node will use an IPv6 address that we have already configured. If the user has specified a non-null value for *dev6AddrNotifyFuncPtr* and the Treck stack detects that another node is using an IPv6 address that we have already configured, then the user is notified of the TM\_6\_DEV\_ADDR\_DUP\_DETECTED event. It is up to the user to specify what recovery, if any, is performed ([RFC2462].R5.4.4:10).

For example, when you are notified of the TM\_6\_DEV\_ADDR\_DUP\_DETECTED event, you could unconfigure the conflicting IPv6 address by calling **tfNgUnConfigInterface**. However, this would impact any applications which are currently using that IPv6 address. Note: if the only link-local scope IPv6 address on the interface is unconfigured, the IPv6 part of the interface will be disabled.

If you are notified of the event TM\_6\_DEV\_ADDR\_CONFIG\_FAILED for a link-local scope IPv6 address, this likely indicates that the IPv6 part of the interface was disabled due to an interface ID conflict. In which case calling **tfCheckOpenInterface** with *protocolFamily* set to PF\_INET6 returns TM\_EPERM. Recovery usually entails either setting a different interface ID by calling **tf6Eui64SetInterfaceId** and then attempting to restart the IPv6 part of the interface by calling **tfNgOpenInterface** specifying an IPv6 address of all 0's (i.e. the IPv6 unspecified address), or manually configuring a link-local IPv6 address on the interface. However, it is possible that you get this failure because your network controller hardware does not support filtering of its own multicasts, but instead does loopback of its own multicasts which would cause Duplicate Address Detection to always fail. If this is the case, then specifying the IPv6-specific device flag TM\_6\_DEV\_MCAST\_HW\_LOOPBACK in *ipv6Flags* should fix the problem. Note that generally you want to avoid setting this device flag since it causes Duplicate Address Detection to not be as robust.

The function prototype for the user-defined notify function (specified by *dev6AddrNotifyFuncPtr*) is:

```
void myDev6AddrNotifyFunc(
    ttUserInterface interfaceHandle,
    unsigned int multiHomeIndex,
    struct sockaddr_storage *ipv6AddrPtr,
    int event );
```

where *event* is the IPv6-specific address configuration event (i.e. TM\_6\_DEV\_ADDR\_CONFIG\_STARTED, ...) for the specified IPv6 address (i.e. *ipv6AddrPtr*) at the specified multi-home index (i.e. *multiHomeIndex*) on the specified interface (i.e. *interfaceHandle*).

By default, Duplicate Address Detection will be performed for every IPv6 address configured on the interface. IPv6 addresses which are auto-configured using the interface ID all share the same low-order 64 bits, set to the interface ID. Specifying the IPv6-specific device flag `TM_6_DEV_OPTIMIZE_DAD` causes Duplicate Address Detection to only be performed on the link-local scope address for the set of addresses that are auto-configured using the interface ID ([RFC2462].R5.4:30).

#### New Parameters:

Parameter	Description
<i>prefixLen</i>	The length (in bits) of the subnet prefix part of the address specified by <i>ipAddrPtr</i> . For an IPv6 address, the value specified must be in the range of 4 to 128, and will typically be 64. For an IPv4 address, the value specified is the number of most significant bits in the associated IPv4 netmask that are set to 1.
<i>flags</i>	General device flags and IPv4-specific interface flags, same as the <i>flag</i> parameter passed to <b>tfConfigInterface</b> .
<i>ipv6Flags</i>	IPv6-specific interface/device flags: <code>TM_6_DEV_IP_FORW_ENB</code> <code>TM_6_DEV_IP_FORW_MCAST_ENB</code> <code>TM_6_DEV_MCAST_HW_LOOPBACK</code> <code>TM_6_DEV_OPTIMIZE_DAD</code> <code>TM_6_USE_AUTO_IID</code>

Only device flags that start with “`TM_6_DEV_`” are set in *ipv6Flags*, all others are set in *flags*.

<i>dev6AddrNotifyFuncPtr</i>	The function to call to notify the user of IPv6-specific address configuration events, or <code>TM_6_DEV_ADDR_NOTIFY_FUNC_NULL_PTR</code> if notification is not desired. The events are: <code>TM_6_DEV_ADDR_CONFIG_STARTED</code> <code>TM_6_DEV_ADDR_CONFIG_FAILED</code> <code>TM_6_DEV_ADDR_CONFIG_COMPLETE</code> <code>TM_6_DEV_ADDR_DUP_DETECTED</code> <code>TM_6_DEV_ADDR_DEPRECATED</code> <code>TM_6_DEV_ADDR_INVALIDATED</code>
------------------------------	---

#### New Return Codes:

Value	Meaning
<code>TM_EINVAL</code>	<i>multiHomeIndex</i> was greater than or equal to <code>TM_MAX_IPS_PER_IF</code> or invalid value specified for <i>prefixLen</i> .
<code>TM_EAFNOSUPPORT</code>	<i>ipAddrPtr-&gt;addrFamily</i> was set to an invalid value for address family; valid values are <code>AF_INET</code> and <code>AF_INET6</code> .
<code>TM_EPERM</code>	You attempted to open the interface for IPv6, however there was no way to configure a link-local scope IPv6 address on the interface. Typically, this happens when you use the null link-layer, and you don't call <b>tf6Eui64SetInterfaceId</b> to set the interface ID before attempting to open the interface for IPv6.
<code>TM_ENOSPC</code>	Failed to configure an IPv4-compatible IPv6 because a different IPv4 address was already configured in the IPv4 address list at the specified multi-home index.
<code>TM_EINPROGRESS</code>	Configuration has not completed for the specified multi-home index. If you are configuring an IPv6 address on the interface, then this error code could indicate that Duplicate Address Detection is still being performed.

#### 4.4.11 `tfNgGetIpAddress`

```

int                tfNgGetIpAddress
(
ttUserInterface   interfaceHandle,
struct sockaddr_storage * ifIpAddressPtr,
int               addrFamily,
unsigned int      multiHomeIndex
);

```

##### Function Description

Next Generation API replaces `tfGetIpAddress`.

When *addrFamily* is set to `AF_INET4`, this API may be used to retrieve each IPv4 address manually configured on the interface by the user.

When *addrFamily* is set to `AF_INET6`, this API may be used to retrieve auto-configured IPv6 addresses, as well as IPv6 addresses manually configured on the interface by the user. To retrieve auto-configured IPv6 addresses, specify a value for *multiHomeIndex* which is greater than or equal to `TM_MAX_IPS_PER_IF` and less than `(TM_MAX_IPS_PER_IF + TM_6_MAX_AUTOCONF_IPS_PER_IF)`, since auto-configured IPv6 addresses are stored after manually configured IPv6 addresses on the interface. Note that the macros `TM_MAX_IPS_PER_IF` and `TM_6_MAX_AUTOCONF_IPS_PER_IF` are not publicly accessible, unless you explicitly `#define` them in your `trsystem.h` file.

##### New Parameters

Parameter	Description
<i>addrFamily</i>	The address family (i.e. <code>AF_INET</code> for IPv4, <code>AF_INET6</code> for IPv6) of the address that you want to get.

##### New Return Codes

Value	Meaning
<code>TM_EAFNOSUPPORT</code>	<i>addrFamily</i> was set to an invalid value for address family; valid values are <code>AF_INET</code> and <code>AF_INET6</code> .

#### 4.4.12 `tfNgGetPrefixLen`

```
int          tfNgGetPrefixLen
(
ttUserInterface  interfaceHandle,
ttUser8Bit *    prefixLenPtr,
int             addrFamily,
unsigned int    multiHomeIndex
);
```

#### Function Description

Next Generation API replaces `tfGetNetMask`.

When *addrFamily* is set to `AF_INET4`, this API may be used to retrieve the prefix length for each IPv4 address manually configured on the interface by the user.

When *addrFamily* is set to `AF_INET6`, this API may be used to retrieve the prefix length for auto-configured IPv6 addresses, as well as IPv6 addresses manually configured on the interface by the user. See the description of `tfNgGetIpAddress` for information on what values of *multiHomeIndex* correspond to auto-configured IPv6 addresses.

#### New Parameters:

Parameter	Description
<i>prefixLenPtr</i>	Pointer to the prefix length of the IP address specified by <i>addrFamily</i> and <i>multiHomeIndex</i> . The prefix length is the length (in bits) of the subnet prefix part of the IP address. For an IPv6 address, the prefix length is in the range of 0 to 128, and will typically be 64. For an IPv4 address, the prefix length is the number of most significant bits in the associated IPv4 netmask that are set to 1. The memory that this points to must be allocated by the caller, i.e. a local variable in the caller's address space.
<i>addrFamily</i>	The address family (i.e. <code>AF_INET</code> for IPv4, <code>AF_INET6</code> for IPv6) of the address that you want to get the prefix length for.

#### New Return Codes:

Value	Meaning
<code>TM_EINVAL</code>	Invalid value specified for <i>prefixLenPtr</i> .
<code>TM_EAFNOSUPPORT</code>	<i>addrFamily</i> was set to an invalid value for address family; valid values are <code>AF_INET</code> and <code>AF_INET6</code> .

### 4.4.13 tfNgOpenInterface

```

int                tfNgOpenInterface
(
ttUserInterface   interfaceHandle,
const struct sockaddr_storage * ipAddrPtr,
int               prefixLen,
int               flags,
int               ipv6Flags,
int               buffersPerFrameCount,
tt6DevAddrNotifyFuncPtr dev6AddrNotifyFuncPtr
);

```

#### Function Description

Next Generation API replaces **tfOpenInterface**. See the description of **tfNgConfigInterface**.

#### New Parameters:

##### Parameter

##### Description

*prefixLen*

The length (in bits) of the subnet prefix part of the address specified by *ipAddrPtr*. For an IPv6 address, the value specified must be in the range of 4 to 128, and will typically be 64. For an IPv4 address, the value specified is the number of most significant bits in the associated IPv4 netmask that are set to 1.

*flags*

General device flags and IPv4-specific interface flags, same as the *flag* parameter passed to **tfConfigInterface**.

*ipv6Flags*

IPv6-specific device and interface flags:

```

TM_6_DEV_IP_FORW_ENB
TM_6_DEV_IP_FORW_MCAST_ENB
TM_6_DEV_MCAST_HW_LOOPBACK
TM_6_DEV_OPTIMIZE_DAD
TM_6_DEV_USE_AUTO_IID

```

*dev6AddrNotifyFuncPtr*

The function to call to notify the user of IPv6-specific address configuration events, or `TM_6_DEV_ADDR_NOTIFY_FUNC_NULL_PTR` if notification is not desired. The events are:

```

TM_6_DEV_ADDR_CONFIG_STARTED
TM_6_DEV_ADDR_CONFIG_FAILED
TM_6_DEV_ADDR_CONFIG_COMPLETE
TM_6_DEV_ADDR_DUP_DETECTED
TM_6_DEV_ADDR_DEPRECATED
TM_6_DEV_ADDR_INVALIDATED

```

See the description of **tfNgConfigInterface** for more information.

**New Return Codes:**

<b>Value</b>	<b>Meaning</b>
TM_EINVAL	Invalid value specified for <i>prefixLen</i> .
TM_EAFNOSUPPORT	<i>ipAddrPtr-&gt;addrFamily</i> was set to an invalid value for address family; valid values are AF_INET and AF_INET6.
TM_EPERM	You attempted to open the interface for IPv6, however there was no way to configure a link-local scope IPv6 address on the interface. Typically, this happens when you use the null link-layer, and you didn't call <b>tf6Eui64SetInterfaceId</b> to set the interface ID before attempting to open the interface for IPv6.
TM_ENOSPC	Failed to configure an IPv4-compatible IPv6 because a different IPv4 address was already configured in the IPv4 address list at the specified multi-home index.
TM_EINPROGRESS	Configuration has not completed for the specified multi-home index. If you are configuring an IPv6 address on the interface, then this error code could indicate that Duplicate Address Detection is still being performed.

**4.4.13.1 tfNgOpenInterface Example**

```

#include <trsocket.h>

void main (void)
{
    ttUserLinkLayer          linkLayerHandle;
    ttUserInterface         interfaceHandle;
    struct sockaddr_storage  ipNgAddr;
    ttUserIpAddress         mask;

    ...
    /* Start Treck */
        tfStartTreck();

    /* Use the Ethernet Link Layer */
        linkLayerHandle = tfUseEthernet();

    /* Add the Interface */
        interfaceHandle=tfAddInterface(
            "MPC860.SCC1",
            linkLayerHandle,
            tfMpc860EtherOpen,
            tfMpc860EtherClose,
            tfMpc860EtherSend,
            tfMpc860EtherReceive,
            0,
            tfMpc860Ioctl,
            tfMpc860GetPhyAddr,
            &errorCode);

    /* store our IPv4 address as type sockaddr_storage for use with the
     * "Next Generation" APIs
     */
        tfMemSet(&ipNgAddr, 0, sizeof(struct sockaddr_storage));
        ipNgAddr.ss_family = AF_INET; /* IPv4 address */
        ipNgAddr.ss_len = sizeof(struct sockaddr_storage);
        ipNgAddr.addr.ipv4.sin_addr.s_addr = inet_addr("192.168.0.2");
        mask = inet_addr("255.255.255.0");

    /* open the interface for use with IPv4 */
        errorCode=tfNgOpenInterface(
            interfaceHandle,
            &ipNgAddr,

```

```
tf4NetmaskToPrefixLen(mask),
configFlags,
0, /* IPv6-specific flags */
scatteredBufferCount,
TM_6_DEV_ADDR_NOTIFY_FUNC_NULL_PTR);

/* open the interface for use with IPv6:
 * start IPv6 stateless address auto-configuration to get a link-local
 * scope IPv6 address assigned to the interface.
 */
tfMemSet(&ipNgAddr, 0, sizeof(struct sockaddr_storage));
ipNgAddr.ss_family = AF_INET6; /* IPv6 address */
ipNgAddr.ss_len = sizeof(struct sockaddr_storage);

errorCode=tfNgOpenInterface(
    interfaceHandle,
    &ipNgAddr, /* IPv6 address is all 0's, auto-configuration */
    0, /* prefix length does not matter for auto-configuration */
    configFlags,
    0, /* IPv6-specific flags */
    scatteredBufferCount,
    TM_6_DEV_ADDR_NOTIFY_FUNC_NULL_PTR);

/* wait for auto-configuration to complete */
while (tfCheckOpenInterface(interfaceHandle, AF_INET6)
    != TM_ENOERROR)
{
    OSTimeDly(1);
}

...
```

#### 4.4.14 **tfNgUnConfigInterface**

```

int                tfNgUnConfigInterface
(
ttUserInterface  interfaceHandle,
int              addrFamily,
unsigned int     multiHomeIndex
);

```

##### Function Description

Next Generation API replaces **tfUnConfigInterface**.

When *addrFamily* is set to AF\_INET4, this API may be used to unconfigure an IPv4 address manually configured on the interface by the user. When *addrFamily* is set to AF\_INET6, this API may be used to unconfigure an auto-configured IPv6 address, as well as IPv6 addresses manually configured on the interface by the user. See the description of **tfNgGetIpAddress** for information on what values of *multiHomeIndex* correspond to auto-configured IPv6 addresses. When the Treck stack is running in dual IP layer mode, if there is an IPv4-compatible IPv6 address configured in the list of multi-homed IPv6 addresses on the interface at the specified *multiHomeIndex*, and the corresponding embedded IPv4 address is configured at the same *multiHomeIndex* in the list of multi-homed IPv4 addresses on the interface, then both the IPv4-compatible IPv6 address and the embedded IPv4 address are removed from the interface.

##### New Parameters:

Parameter	Description
<i>addrFamily</i>	The address family (i.e. AF_INET for IPv4, AF_INET6 for IPv6) of the address that you want to remove from the interface.

##### New Return Codes:

Value	Meaning
TM_EAFNOSUPPORT	<i>addrFamily</i> was not set to either AF_INET or AF_INET6.

## 4.5 ARP / Routing Table API

### 4.5.1 tf6AddDefaultGatewayTunnel

```

int                                     tf6AddDefaultGatewayTunnel
(
const struct sockaddr_storage TM_FAR *  endpointIpAddrPtr,
int                                       flags
);

```

#### Function Description

*This function only supports IPv6.*

Create a default gateway tunnel for running IPv6 over IPv4. A host that is on a link without any IPv6 routers but which can reach an IPv6 router via the existing IPv4 routing infrastructure (i.e. the IPv6 router is reachable via an IPv4 address, specified by *endpointIpAddrPtr*) can use this API to set this IPv6 router as the default gateway for all IPv6-capable interfaces. The configured tunnel created by this function is used to reach this default gateway (i.e. over IPv4), and operates as the IPv6 default route, meaning that it will only be used to route an IPv6 packet if there are no other routes that match the destination IP address of the packet. ([RFC2893].R4.1:10, [RFC2893].R4:10)

#### Parameters

Parameter	Description
<i>endpointIpAddrPtr</i>	Pointer to the tunnel endpoint IP address, which is the IPv4 address of the IPv6 router. The memory that this points to must be allocated by the caller, i.e. a local variable in the caller's address space.
<i>flags</i>	Not currently used.

#### Returns

Value	Meaning
TM_EINVAL	One of the parameters was invalid.
TM_EAFNOSUPPORT	The specified tunnel endpoint address was not an IPv4-formatted address, i.e. <i>endpointIpAddrPtr-&gt;addrFamily</i> was not set to AF_INET.
TM_EHOSTUNREACH	The tunnel endpoint (i.e. IPv6 router) was unreachable.
TM_EALREADY	A default gateway tunnel has already been configured.

### 4.5.2 tf6DelDefaultGatewayTunnel

```
int          tf6DelDefaultGatewayTunnel
(
          void
);
```

#### Function Description

*This function only supports IPv6.*

Delete/unconfigure the default gateway tunnel, if one exists.

#### Parameters

None

#### Returns

Value	Meaning
TM_ENOERROR	Success.
TM_EINVAL	One of the parameters was invalid.
TM_ENOENT	No default gateway tunnel with the specified endpoint address.
TM_EAFNOSUPPORT	The specified tunnel endpoint address was not an IPv4-formatted address, i.e. <i>endpointIpAddrPtr-&gt;addrFamily</i> was not set to AF_INET.

### 4.5.3 tf6GetDefaultGatewayTunnel

```
int                                tf6GetDefaultGatewayTunnel
(
struct sockaddr_storage TM_FAR *  endpointIpAddrPtr
);
```

#### Function Description

*This function only supports IPv6.*

This function gets the endpoint IP address of the default gateway tunnel into the user variable pointed to by *endpointIpAddrPtr*. If there is no default gateway tunnel configured, then this function returns TM\_ENOENT.

#### Parameters

Parameter	Description
<i>endpointIpAddrPtr</i>	When this function returns TM_ENOERROR, then the memory that <i>endpointIpAddrPtr</i> points to is set to the tunnel endpoint IP address, which is the IPv4 address of the IPv6 router. The memory that this points to must be allocated by the caller, i.e. a local variable in the caller's address space.

#### Returns

Value	Meaning
TM_ENOERROR	Default gateway tunnel was found – IP address copied into <i>endpointIpAddrPtr</i> .
TM_EINVAL	<i>endpointIpAddrPtr</i> is null.
TM_ENOENT	Default gateway tunnel not found.

### 4.5.4 tf6GetPathMtu

```
int                                tf6GetPathMtu
(
sockaddr_storage *destIpAddrPtr,
int TM_FAR *      pathMtuPtr
);
```

#### Function Description

Retrieves the current Path MTU for the specified destination address. This path MTU does *not* take into account overhead added by IPSec, but does include protocol tunneling overhead. [IPV6REQ].R2.12:10

#### Parameters

Parameter	Description
<i>destIpAddrPtr</i>	IPv6 address of the destination to get the Path MTU for.
<i>pathMtuPtr</i>	Set to the current IPv6 Path MTU upon return, if no error occurred.

#### Returns

Value	Meaning
TM_ENOERROR	IPv6 Path MTU successfully returned.
TM_EINVAL	Destination address is not an IPv6 address.
TM_EINVAL	Destination address pointer is null.
TM_EINVAL	Pointer to Path MTU result is null.
TM_ENOENT	No route to the destination IPv6 address was found.

### 4.5.5 **tf6RegisterIpForwCB**

```
int                tf6RegisterIpForwCB
(
  tt6UserIpForwCBFuncPtr  ipForwCBFuncPtr
);
```

#### **Function Description**

Used to register a function for the Treck stack to call when an IPv6 packet cannot be forwarded. Refer to the IPv4-specific API **tfRegisterIpForwCB**.

```
typedef int (TM_CODE_FAR * tt6UserIpForwCBFuncPtr)(
    const struct in6_addr TM_FAR * srcIpAddrPtr,
    const struct in6_addr TM_FAR * destIpAddrPtr);
```

### 4.5.6 **tfNgAddArpEntry**

```
int                tfNgAddArpEntry
(
  const struct sockaddr_storage * arpIpAddrPtr,
  const ttUser8Bit *             physAddrPtr,
  int                             physAddrLen
);
```

#### **Function Description**

Next Generation API replaces **tfAddArpEntry**.

### 4.5.7 tfNgAddStaticRoute

```

int                tfNgAddStaticRoute
(
  ttUserInterface  interfaceHandle,
  const struct sockaddr_storage * destIpAddrPtr,
  int              prefixLen,
  const struct sockaddr_storage * gatewayPtr,
  int              hops
);

```

#### Function Description

Next Generation API replaces **tfAddStaticRoute**.

#### New Parameters:

Parameter	Description
<i>prefixLen</i>	The length (in bits) of the subnet prefix part of the address specified by <i>destIpAddrPtr</i> . For an IPv6 address, the value specified must be in the range of 0 to 128, and will typically be 64. For an IPv4 address, the value specified is the number of most significant bits in the associated IPv4 netmask that are set to 1.

#### New Return Codes:

Value	Meaning
TM_EINVAL	Invalid value specified for <i>prefixLen</i> .
TM_EAFNOSUPPORT	<i>destIpAddrPtr-&gt;addrFamily</i> or <i>gatewayPtr-&gt;addrFamily</i> was set to an invalid value for address family; valid values are AF_INET and AF_INET6.

### 4.5.8 tfNgDelArpEntryByIpAddr

```

int                tfNgDelArpEntryByIpAddr
(
  const struct sockaddr_storage * arpIpAddrPtr
);

```

#### Function Description

Next Generation API replaces **tfDelArpEntryByIpAddr**.

### 4.5.9 `tfNgDelArpEntryByPhysAddr`

```
int          tfNgDelArpEntryByPhysAddr
(
int          addrFamily,
const ttUser8Bit * physAddrPtr,
int          physAddrLen
);
```

#### Function Description

Next Generation API replaces `tfDelArpEntryByPhysAddr`.

#### New Parameters:

Parameter	Description
<i>addrFamily</i>	Set to <code>AF_INET</code> if you want to delete an entry from the IPv4 ARP Cache, <code>AF_INET6</code> if you want to delete an entry from the IPv6 Neighbor Cache.

#### New Return Codes:

Value	Meaning
<code>TM_EAFNOSUPPORT</code>	<i>addrFamily</i> was not set to either <code>AF_INET</code> or <code>AF_INET6</code> .

#### 4.5.10 tfNgDelStaticRoute

```

int                tfNgDelStaticRoute
(
const struct sockaddr_storage * destIpAddrPtr,
int                    prefixLen
);

```

##### Function Description

Next Generation API replaces **tfDelStaticRoute**.

##### New Parameters:

###### Parameter

*prefixLen*

###### Description

The length (in bits) of the subnet prefix part of the address specified by *destIpAddrPtr*. For an IPv6 address, the value specified must be in the range of 0 to 128, and will typically be 64. For an IPv4 address, the value specified is the number of most significant bits in the associated IPv4 netmask that are set to 1.

##### New Return Codes:

###### Value

TM\_EINVAL

TM\_EAFNOSUPPORT

###### Meaning

Invalid value specified for *prefixLen*.

*destIpAddrPtr->addrFamily* was set to an invalid value for address family; valid values are AF\_INET and AF\_INET6.

### 4.5.11 tfNgDisablePathMtuDisc

```
int                                     tfNgDisablePathMtuDisc
(
sockaddr_storage TM_FAR * destIpAddrPtr,
int                                     pathMtu
);
```

**Function Description**

Disables Path MTU discovery and sets the path MTU for the IPv4 or IPv6 destination address to the specified value. ([RFC1981].R5.6:10)

**Parameters**

<b>Parameter</b>	<b>Description</b>
destIpAddrPtr	Pointer to the destination address to set the path MTU value for.
pathMtu	New path MTU value for the specified destination.

**Returns**

<b>Value</b>	<b>Meaning</b>
TM_ENOERROR	Path MTU successfully updated.
TM_ENOPROTOOPT	Path MTU discovery not enabled for the specified address type.
TM_EINVAL	Destination address pointer is null.
TM_EINVAL	Destination address is zero.
TM_EPRNOSUPPORT	Destination address is not either IPv6 or IPv4.

#### 4.5.12 tfNgGetArpEntryByIpAddr

```
int                tfNgGetArpEntryByIpAddr
(
const struct sockaddr_storage *arpIpAddrPtr,
ttUser8Bit *        physAddrPtr,
int                 physAddrBufLen,
int *               physAddrLenPtr
);
```

##### Function Description

Next Generation API replaces `tfGetArpEntryByIpAddr`.

##### New Parameters:

Parameter	Description
<i>physAddrBufLen</i>	The length in bytes of the buffer pointed to by <i>physAddrPtr</i> .
<i>physAddrLenPtr</i>	Output parameter, set to the length of the physical address returned in the buffer pointed to by <i>physAddrPtr</i> .

### 4.5.13 tfNgGetArpEntryByPhysAddr

```

int          tfNgGetArpEntryByPhysAddr
(
int          addrFamily,
const ttUser8Bit * physAddrPtr,
int          physAddrLen,
struct sockaddr_storage * arpIpAddrPtr
);
    
```

**Function Description**

Next Generation API replaces **tfGetArpEntryByPhysAddr**.

**New Parameters:**

<b>Parameter</b>	<b>Description</b>
<i>addrFamily</i>	Set to AF_INET if you want to get an entry from the IPv4 ARP Cache, AF_INET6 if you want to get an entry from the IPv6 Neighbor Cache.

**New Return Codes:**

<b>Value</b>	<b>Meaning</b>
TM_EAFNOSUPPORT	<i>addrFamily</i> was not set to either AF_INET or AF_INET6.

#### 4.5.14 tfSetReachable

```

int                tfSetReachable
(
struct sockaddr_storage *    addrPtr
ttUserInterface    interface
int *                reachableMsecPtr
);

```

##### Function Description

Applications call this function periodically to notify Neighbor Unreachability Detection of the reachability of a peer node. Essentially, this allows applications to “refresh” the ARP cache entry that is used when sending to the peer. ([RFC2461].R7.3.1:10, [RFC2461].R7.3.1:20, [RFC2461].R7.3.1:30, [RFC2461].R7.3.1:40)

##### Parameters

Parameter	Description
<i>addrPtr</i>	IP address of a peer node that the application has received reachability confirmation for in the form of “forward progress”; it could be a neighbor, or a remote node reachable through router(s). If the peer node is a neighbor, the reachability for the neighbor will be set to REACHABLE, otherwise the reachability for the first-hop router in the path to the peer node is set to REACHABLE.([RFC2461].R7.3.1:30, [RFC2461].R7.3.1:40)
<i>interface</i>	Only need when <i>addrPtr</i> is an IPv6 local scope address. This identifies the outgoing interface that the address specified by <i>addrPtr</i> is reachable through.
<i>reachableMsecPtr</i>	Output parameter; if not NULL, then the new reachable time (in milliseconds) of the matching ARP cache entry will be stored here on return.

##### Returns

Return value	Description
TM_ENOERROR	Success
TM_EINVAL	<i>addrPtr</i> was NULL, or <i>addrPtr</i> points to a local scope IPv6 address but <i>interface</i> was invalid.
TM_ENOENT	No ARP entry exists for the specified peer IP address; send some packets first to the peer to cause the ARP entry to be created.

## 4.6 PPP Link Layer API

### 4.6.1 Using PPP in dual IPv4/IPv6 mode

PPP supports IPv6 with an additional Network Control Protocol (NCP), IPV6CP. Negotiation for IPV6CP may take before, after or concurrent with IPCP negotiation (for IPv4). It is possible for a certain connection only one NCP will fail; for instance, IPV6CP may fail if the peer does not support IPv6.

After an NCP fails, the stack does not close the serial connection immediately. Rather, it waits some amount of time (as specified by the `TM_PPP_OPTION_TIMEOUT` parameter – please see above). This time allows you to start another NCP if the first one fails, by calling `tfNgConfigInterface`. However, if `TM_PPP_OPTION_TIMEOUT` expires or the second NCP also fails, the connection will be closed.

#### Using PPP in dual IPv4/IPv6 mode:

To open both IPv4 and IPv6 on a PPP link requires two separate calls to `tfNgOpenInterface`, one for each address family (see example below). The user is notified of the status of an individual address family through the `TM_LL_IPx_OPEN_FAILED` and `TM_LL_IPx_OPEN_COMPLETE` flags.

Both address families may be started concurrently or sequentially. For instance, you may open IPv6 by calling `tfNgOpenInterface` again immediately after calling `tfNgOpenInterface` for IPv4, as in the example below. In this case, negotiation for the network protocols will occur concurrently.

Alternatively, you may wait until one address family fails before opening the second. For instance, you can attempt to open IPv6 on the link through a single call to `tfNgOpenInterface`. If this fails, for instance if the peer does not support IPv6, you will be notified with the `TM_LL_IP6_OPEN_FAILED` flag. At this point, you may want to open IPv4 on the link, which can be done through another call to `tfNgOpenInterface`. After negotiation of the first address family, PPP will delay for some amount of time before completely closing the PPP link allowing you time to start a second protocol family. This amount of time is controlled by the `TM_PPP_OPEN_TIMEOUT` option and defaults to 15 seconds.

#### 4.6.1.1 PPP Dual IPv4/IPv6 Usage Example

```

struct sockaddr_storage configAddr;

pppFlags = 0;

/* Open IPv4 on the PPP interface */
bzero( (void *) &configAddr, sizeof(configAddr) );
configAddr.addr.ipv4.sin_family = PF_INET;
configAddr.addr.ipv4.sin_len    =
    sizeof(struct sockaddr_in);

errorCode =
    tfNgOpenInterface( interfaceHandle,
                       &configAddr,
                       64, /* prefix length (unused
                           for PPP)*/
                       TM_DEV_MCAST_ENB,
                       0,
                       1,
                       (tt6DevAddrNotifyFuncPtr)0 );

/* Open IPv6 on the PPP interface. */

```

```

    bzero( (void *) &configAddr, sizeof(configAddr) );
    configAddr.addr.ipv6.sin6_family = PF_INET6;

    errorCode =
        tfNgOpenInterface( interfaceHandle,
                           &configAddr,
                           64, /* prefix length */
                           TM_DEV_MCAST_ENB,
                           0,
                           1,
                           ipv6AddrNotify );

```

#### 4.6.2 tfNgGetPt2PtPeerIpAddress

```

int                tfNgGetPt2PtPeerIpAddress
(
    ttUserInterface    interfaceHandle,
    struct sockaddr_storage * ifIpAddressPtr
);

```

##### Function Description

Next Generation API replaces **tfGetPppPeerIpAddress**.

#### 4.6.3 tfNgSetPt2PtPeerIpAddress

```

int                tfNgSetPt2PtPeerIpAddress
(
    ttUserInterface    interfaceHandle,
    const struct sockaddr_storage * ifIpAddressPtr
);

```

##### Function Description

Next Generation API replaces **tfSetPppPeerIpAddress**.

#### 4.6.4 tfPppSetOption

*Please see tfPppSetOption in Chapter 7: 'Optional Protocols' of the Treck TCP/IP User's Manual. The following are additional options for IPv6 support.*

TM_PPP_IPV6CP_PROTOCOL protocolLevel		
optionName	Length	Description
TM_IPV6CP_RETRY	1	Set the maximum number of IPV6CP config requests that will be sent without receiving an IPV6CP nak/ack/reject. remoteLocalFlag has no effect. <b>Default: 10</b>
TM_IPV6CP_TIMEOUT	1	Sets the IPV6CP retransmission timeout value (in seconds). remoteLocalFlag has no effect. <b>Default: 1 second</b>
TM_IPV6CP_MAX_FAILURES	1	Sets the maximum number of IPV6CP configuration failures. This determines the maximum number of configuration NAKs that will be sent before we reject an option. <b>Default: 5</b>
TM_IPV6CP_COMP_PROTOCOL	2	Specifies the type of compression to use over the link (optional). TM_PPP_IPHC_PROTOCOL selects RFC-2507 style IP header compression (you must also call <b>tfUseIpHdrComp</b> before setting this option).
TM_IPV6CP_IPHC_TCP_SPACE	2	IP Header Compression: The maximum number of slots used to store TCP header compression info; this value is determined by the maximum number of concurrent TCP sessions that are expected over this link. <b>Default: 4</b>
TM_IPV6CP_IPHC_NON_TCP_SPACE	2	IP Header Compression: The maximum number of slots used to store non-TCP (UDP, etc) header compression info; this value is determined by the maximum number of concurrent non-TCP sessions that are expected across this link. <b>Default: 4</b>
TM_IPV6CP_IPHC_MAX_PERIOD	2	IP Header Compression: The maximum interval between sending full headers. <b>Default: 256</b>
TM_IPV6CP_IPHC_MAX_TIME	2	IP Header Compression: The maximum time interval, in seconds, between sending full headers. <b>Default: 5 seconds</b>
TM_IPV6CP_IPHC_MAX_HEADER	2	IP Header Compression: The largest header size in octets that may be compressed. <b>Default: 168 bytes</b>
TM_PPP_PROTOCOL protocolLevel		
optionName	Length	Description
TM_PPP_OPEN_TIMEOUT	1	The amount of time in seconds PPP will wait after IPCP or IPV6CP fails to open before closing the link. This is only valid when using the stack in dual IPv4/IPv6 mode. Please see below for more details. <b>Default: 15 seconds</b>

The following options should be modified in **tfPppSetOption**:

protocolLevel	optionName	Description of Change
TM_PPP_LCP_PROTOCOL	TM_LCP_MAX_RECV_UNIT	When using PPP on an IPv6 interface, MRU must not be set to less 1280. [RFC2460].R5:205

#### 4.6.5 `tfUseAsyncPpp`

```
#include <trsocket.h>

ttUserLinkLayer          tfUseAsyncPpp
(
ttUserLnkNotifyFuncPtr  linkNotifyFuncPtr
);
```

#### Function Description

This function is used to initialize the asynchronous PPP client link layer. When link up or link down events occur, the stack will call the function passed in. If you do not need notification of events then the parameter should be set to `TM_LL_NOTIFY_FUNC_NULL_PTR`.

Your prototype for your notification function should look like this:

```
void myPppNotifyFunction(ttUserInterface interfaceHandle,
                        int flags);
```

This function is called with the interface handle and a flag. The flag is set to one of the following:

<code>TM_LL_OPEN_COMPLETE:</code>	The PPP link is now ready to accept data from the user. When used in dual IPv4/IPv6 mode, this notification will be sent for the <i>first</i> network layer that is configured.
<code>TM_LL_CLOSE_STARTED</code>	PPP has started to terminate the connection.
<code>TM_LL_CLOSE_COMPLETE</code>	The PPP link is complete closed (for both IPv4 and IPv6).
<code>TM_LL_LCP_UP</code>	LCP negotiation has completed.
<code>TM_LL_PAP_UP</code>	PAP authentication has completed.
<code>TM_LL_CHAP_UP</code>	CHAP authentication has completed.
<code>TM_LL_LQM_UP</code>	LQM is enabled on the link.
<code>TM_LL_LQM_DISABLED</code>	LQM is disabled on the link.
<code>TM_LL_LQM_LINK_BAD</code>	Link quality is bad, user recovery should be attempted.
<code>TM_LL_IP4_OPEN_FAILED</code>	PPP has failed to start IPv4 on the link (IPCP negotiation failed). When using dual IPv4/IPv6 mode, it is still possible to start IPv6 at this point.
<code>TM_LL_IP6_OPEN_FAILED</code>	PPP has failed to start IPv6 on the link (IPV6CP negotiation failed). When using dual IPv4/IPv6 mode, it is still possible to start IPv4 at this point.
<code>TM_LL_OPEN_FAILED</code>	PPP was unable to start either IPv4 or IPv6 on the link. The PPP connection is completely closed.
<code>TM_LL_IP4_OPEN_COMPLETE</code>	IPv4 may now be used on the PPP link (IPCP negotiation was successful).
<code>TM_LL_IP6_OPEN_COMPLETE</code>	IPv6 may now be used on the PPP link (IPV6CP negotiation was successful).

These events may be used to monitor the status of the PPP connection. The PPP connection should not be used before a `TM_LL_OPEN_COMPLETE` event is received, and the device should not be restarted (after a **tfCloseInterface**) before a `TM_LL_CLOSE_COMPLETE` event is received.

From these events, it is also possible to determine why PPP negotiation failed. For instance, if authentication fails, a `TM_LL_LCP_UP` event is first received indicating that the physical link has been negotiated. However, if authentication fails, the next event will be `TM_LL_CLOSE_STARTED` and then `TM_LL_CLOSE_COMPLETE` since the link must be closed if negotiation fails. If authentication is successful the events that are received are `TM_LL_LCP_UP`, `TM_LL_PAP_UP` or `TM_LL_CHAP_UP` and then `TM_LL_OPEN_COMPLETE`.

**Parameters**

**Parameter**

*linkNotifyFuncPtr*

**Description**

The function to call to notify PPP events or  
TM\_LNK\_NOTIFY\_FUNC\_NULL\_PTR if notification is not needed

The events are:

- TM\_LL\_OPEN\_COMPLETE
- TM\_LL\_CLOSE\_STARTED
- TM\_LL\_CLOSE\_COMPLETE
- TM\_LL\_LCP\_UP
- TM\_LL\_PAP\_UP
- TM\_LL\_CHAP\_UP

**Returns**

The PPP Client link layer handle or NULL if there is an error

#### 4.6.6 `tfUseAsyncServerPpp`

```
#include <trsocket.h>

ttUserLinkLayer          tfUseAsyncServerPpp
(
  ttUserLnkNotifyFuncPtr  linkNotifyFuncPtr
);
```

#### Function Description

This function is used to initialize the asynchronous PPP server link layer. When link up or link down events occur, the stack will call the function passed in. If you do not need notification of events then the parameter should be set to `TM_LL_NOTIFY_FUNC_NULL_PTR`.

Your prototype for your notification function should look like this:

```
void myPppNotifyFunction(ttUserInterface interfaceHandle,
                        int flags);
```

This function is called with the interface handle and a flag. The flag is set to one of the following:

<code>TM_LL_OPEN_COMPLETE:</code>	The PPP link is now ready to accept data from the user. When used in dual IPv4/IPv6 mode, this notification will be sent for the <i>first</i> network layer that is configured.
<code>TM_LL_CLOSE_STARTED</code>	PPP has started to terminate the connection.
<code>TM_LL_CLOSE_COMPLETE</code>	The PPP link is complete closed (for both IPv4 and IPv6).
<code>TM_LL_LCP_UP</code>	LCP negotiation has completed.
<code>TM_LL_PAP_UP</code>	PAP authentication has completed.
<code>TM_LL_CHAP_UP</code>	CHAP authentication has completed.
<code>TM_LL_LQM_UP</code>	LQM is enabled on the link.
<code>TM_LL_LQM_DISABLED</code>	LQM is disabled on the link.
<code>TM_LL_LQM_LINK_BAD</code>	Link quality is bad, user recovery should be attempted.
<code>TM_LL_IP4_OPEN_FAILED</code>	PPP has failed to start IPv4 on the link (IPCP negotiation failed). When using dual IPv4/IPv6 mode, it is still possible to start IPv6 at this point.
<code>TM_LL_IP6_OPEN_FAILED</code>	PPP has failed to start IPv6 on the link (IPV6CP negotiation failed). When using dual IPv4/IPv6 mode, it is still possible to start IPv4 at this point.
<code>TM_LL_OPEN_FAILED</code>	PPP was unable to start either IPv4 or IPv6 on the link. The PPP connection is completely closed.
<code>TM_LL_IP4_OPEN_COMPLETE</code>	IPv4 may now be used on the PPP link (IPCP negotiation was successful).
<code>TM_LL_IP6_OPEN_COMPLETE</code>	IPv6 may now be used on the PPP link (IPV6CP negotiation was successful).

These events may be used to monitor the status of the PPP connection. The PPP connection should not be used before a `TM_LL_OPEN_COMPLETE` event is received, and the device should not be restarted (after a **tfCloseInterface**) before a `TM_LL_CLOSE_COMPLETE` event is received.

From these events, it is also possible to determine why PPP negotiation failed. For instance, if authentication fails, a `TM_LL_LCP_UP` event is first received indicating that the physical link has been negotiated. However, if authentication fails, the next event will be `TM_LL_CLOSE_STARTED` and then `TM_LL_CLOSE_COMPLETE` since the link must be closed if negotiation fails. If authentication is successful the events that are received are `TM_LL_LCP_UP`, `TM_LL_PAP_UP` or `TM_LL_CHAP_UP` and then `TM_LL_OPEN_COMPLETE`.

## Parameters

### Parameter

*linkNotifyFuncPtr*

### Description

The function to call to notify PPP events or  
TM\_LNK\_NOTIFY\_FUNC\_NULL\_PTR if notification is not needed

The events are:

TM\_LL\_OPEN\_COMPLETE

TM\_LL\_CLOSE\_STARTED

TM\_LL\_CLOSE\_COMPLETE

TM\_LL\_LCP\_UP

TM\_LL\_PAP\_UP

TM\_LL\_CHAP\_UP

## Returns

The PPP Server link layer handle or NULL if there is an error

## 5 Application Reference

### 5.1.1 tfNgDnsSetServer

```

int                                     tfNgDnsSetServer
(
struct sockaddr_storage *             serverAddrPtr,
int                                   serverNumber
);

```

#### Function Description

Sets the address of the primary and secondary DNS server. To set the primary DNS server *serverNumber* should be set to `TM_DNS_PRI_SERVER`; for the secondary server it should be set to `TM_DNS_SEC_SERVER`. To remove a previously set entry, set *serverAddrPtr* to `NULL`.

#### Parameters

##### Parameter

*serverAddrPtr*

*serverNumber*

##### Description

Pointer to the address of the IPv4 or IPv6 address of the specified DNS server.

`TM_DNS_PRI_SERVER` or `TM_DNS_SEC_SERVER`

#### Returns

##### Value

`TM_EINVAL`

##### Meaning

Server number is not `TM_DNS_PRI_SERVER` or `TM_DNS_SEC_SERVER`

### 5.1.2 tfNgFtpConnect

```
int          tfNgFtpConnect (
ttUserFtpHandle  ftpUserHandle,
struct sockaddr_storage *  ipAddrPtr
);
```

#### Function Description

This function replaces **tfFtpConnect** and attempts to connect to a remote FTP server.

#### Parameters

Parameter	Description
<i>ftpUserHandle</i>	FTP session handle
<i>ipAddrPtr</i>	Pointer to a structure containing the address of the address of the remote FTP server

#### Returns

Value	Meaning
TM_ENOERROR	Success
TM_EWOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EALREADY	Command in progress (previous call did not yet finish)
TM_EACCES	Trying to connect to a different FTP server without disconnecting from the current server.
TM_EINVAL	Invalid FTP session pointer.
TM_FTP_SERVREADY	Service ready in 'n' minutes (for exact time, use <b>tfFtpGetReplyText</b> to retrieve full reply text).
TM_FTP_SERVNAVAL	Service not available, closing connection.

### 5.1.3 tfNgFtpdUserStart

```
int          tfNgFtpdUserStart(  
int          fileFlags,  
int          maxConnections,  
int          maxBackLog,  
int          idleTimeout,  
int          blockingState,  
unsigned long flags  
);
```

#### Function Description

This function replaces **tfFtpdUserStart** and opens an FTP server socket and starts listening for incoming connections. **tfFtpdUserStart** can be either blocking or non-blocking as specified by its *blockingState* parameter.

#### New Parameters

*flags*

One or more of the following flags: **TM\_FTPD\_IPV4\_ONLY** – When operating in dual IPv4/IPv6 mode, only allow IPv4 connections to the FTP server. Otherwise, allow both IPv4 and IPv6 connections.

### 5.1.4 tfNgPingOpenStart

```

int                                     tfNgPingOpenStart
(
const struct sockaddr_storage * remoteAddrPtr,
ttUser32Bit                          pingInterval,
int                                    pingDataLength,
ttPingCBFuncPtr                      pingUserCBFuncPtr
);

```

#### Function Description

This function opens an ICMP socket and starts sending PING echo requests to a remote host as specified by the *remoteAddrPtr* parameter. PING echo requests are sent every *pingInterval* milliseconds. The PING length (not including IP and ICMP headers) is given by the *pingDataLength* parameter. If the *pingUserCBFuncPtr* parameter is non-null, the function it points to is called for each received PING echo reply or ICMP error message, with the socket descriptor returned by **tfPingOpenStart** passed as a parameter. To get the PING connection results and statistics, the user must call **tfPingGetStatistics**. To stop the system from sending PING echo re-quests and to close the ICMP socket, the user must call **tfPingClose**.

#### Parameters

Parameter	Description
<i>remoteAddrPtr</i>	Remote IPv4 or IPv6 address to gather ICMP statistics from.
<i>pingInterval</i>	Interval in milliseconds at which to retry sending a Ping packet (if 0, use default of 1 second).
<i>pingDataLength</i>	User Data length of the PING echorequest. If set to zero, defaults to 56bytes. If set to a value between 1, and 3, defaults to 4 bytes.
<i>pingUserCBFuncPtr</i>	Pointer to a user function to becalled upon receiving a networkPING echo reply, or an ICMP errormessage, with the socket descriptoras returned by <b>tfPingOpenStart</b> passed as a parameter. Can be set tonull function pointer if the userdoes not wish to be notified ofincoming network traffic.

#### Returns

New ICMP Socket Descriptor or TM\_SOCKET\_ERROR (-1) on error. If **tfNgPingOpenStart** fails, the *errorCode* can be retrieved with **tfGetSocketError** (TM\_SOCKET\_ERROR):

Value	Meaning
TM_EINVAL	<i>remoteHostNamePtr</i> was a nullpointer
TM_EINVAL	<i>pingInterval</i> was negative
TM_EINVAL	<i>pingDataLength</i> was negative of bigger than 65595, maximum valueallowed by the IP protocol.
TM_ENOBUFS	There was insufficient user memoryavailable to complete the operation.
TM EMSGSIZE	<i>pingDataLength</i> exceeds socketsend queue limit, or <i>pingDataLength</i> exceeds the IPMTU, and fragmentation is notallowed.
TM_EHOSTUNREACH	No route to remote host

**5.1.4.1 tfNgPingOpenStart Example**

```

ttUserInterface interfaceHandle;
struct sockaddr_storage ipNgAddr;

...

tfMemSet(&ipNgAddr, 0, sizeof(struct sockaddr_storage));
ipNgAddr.ss_family = AF_INET6; /* IPv6 address */
ipNgAddr.ss_len = sizeof(struct sockaddr_storage);

/* set the address we want to ping */
errorCode = inet_pton(
    AF_INET6,
    "fe80::240:f4ff:fe52:f537", /* Ping Jin's Linux server */
    &(ipNgAddr.addr.ipv6.sin6_addr));

/* since this is a link-local scope address, we need to set the field
 * sin6_scope_id in sockaddr_in6 to the correct scope ID, so we know
 * which interface to send the ping out on.
 */
errorCode = tf6SockaddrSetScopeId(
    interfaceHandle, &ipNgAddr);

/* start the ping */
pingSd = tfNgPingOpenStart(
    &ipNgAddr,
    1000, /* 1 second between retransmissions */
    100, /* 100 bytes of data */
    (ttPingCBFuncPtr) 0);

if (pingSd != TM_SOCKET_ERROR)
{
    /* let the ping proceed for awhile, yield to the preemptive RTOS */
    OSTimeDly(5000); /* allow other tasks to run for 500 seconds */

    tfPingClose(pingSd); /* stop the ping */
}

```

### 5.1.5 **tfNgTeldOpened**

```
void                tfNgTeldOpen(  
ttUserTeldHandle   teldHandle,  
struct sockaddr_storage * sockAddrPtr  
);
```

Next Generation API replaces **tfNgTeldOpened**.

#### **New parameters:**

<b>Parameter</b>	<b>Description</b>
<i>sockAddrPtr</i>	Pointer to a <code>sockaddr_storage</code> structure containing the IPv4 or IPv6 address of the telnet client.

## 6 Structure Reference

### 6.1.1 addrinfo

```

struct                addrinfo
{
int                  ai_flags;
int                  ai_family;
int                  ai_socktype;
int                  ai_protocol;
unsigned int        ai_addrlen;
char *               ai_canonname;
struct sockaddr *    ai_addr;
struct addrinfo *    ai_next;
};

```

#### Structure Description

Structure used to return address information for a hostname to a user by **getaddrinfo**, and used by a user to specify what sort of addresses should be returned by **getaddrinfo**. Also used internally to store a series of mail host (MX) records on the cache entry.

#### Structure Members

Member	Description
ai_flags	AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST
ai_family	Protocol family (PF_INET or PF_INET6)
ai_socktype	Not used.
ai_protocol	Not used.
ai_addrlen	length of ai_addr
ai_canonname	canonical name for nodename
ai_addr	binary address
ai_next	next structure in linked list

### 6.1.2 in6\_addr

```
struct          in6_addr
{
union {
    u_char      ip6U8[16];
    u_short     ip6U16[8];
    u_long      ip6U32[4];
}
} ip6Addr;
#define s6_addr ip6Addr.ip6U8
```

#### Structure Description

Refer to [RFC2553]. Data structure required by the BSD socket API used to represent an IPv6 address.

#### Structure Members

Member	Description
<i>ip6Addr</i>	128-bit IPv6-formatted address in network byte order.

### 6.1.3 `ipv6_mreq`

```
struct          ipv6_mreq
{
    struct in6_addr  ipv6mr_multiaddr;
    unsigned int     ipv6mr_interface;
};
```

#### Structure Description

Refer to [RFC2553]. Data structure required by the BSD socket API used to join and leave IPv6 multicast groups.

#### Structure Members

Member	Description
<i>ipv6mr_multiaddr</i>	IPv6 multicast address of the multicast group you want to join/leave.
<i>ipv6mr_interface</i>	Interface index of the interface that you want to enable/disable receiving multicasts on for the specified multicast group.

### 6.1.4 sockaddr\_in6

```

struct          sockaddr_in6
{
  u_char        sin6_len;
  u_char        sin6_family;
  u_short       sin6_port;
  u_long        sin6_flowinfo;
  struct in6_addr sin6_addr;
  u_long        sin6_scope_id;
};

```

#### Structure Description

Refer to [RFC2553]. Data structure required by the 4.4BSD socket API used to represent an IPv6 address.

#### Structure Members

Member	Description
<i>sin6_len</i>	Length (in bytes) of the <i>sockaddr_in6</i> structure. The Treck stack sets this field to 24 in any <i>sockaddr_in6</i> structure that it returns to the user application.
<i>sin6_family</i>	Address family, set to AF_INET6.
<i>sin6_port</i>	Transport layer port number.
<i>sin6_flowinfo</i>	This field contains the IPv6 packet header <i>Flow Label</i> as the low-order 20 bits and <i>Traffic Class</i> as the next more significant 8 bits. The high-order 4 bits of this field are reserved. This field allows the application to set the flow label and traffic class associated with the socket via a call to <b>bind</b> or <b>connect</b> . ([RFC2460].R7:10, [IPV6REQ].R2.16:10)
<i>sin6_addr</i>	128-bit IPv6-formatted address in network byte order.
<i>sin6_scope_id</i>	This field is only used for local scope (i.e. link-local, site-local) unicast IPv6 addresses. For global scope unicast IPv6 addresses, as well as for IPv4 addresses, this field is not used and should be set to 0. When the IPv6 address is a link-local scope unicast IPv6 address, then <i>sin6_scope_id</i> must be set to the interface index of the interface associated with that IP address. When the IPv6 address is a site-local scope unicast IPv6 address, then <i>sin6_scope_id</i> must be set to the site identifier of the site (i.e. set of interfaces) associated with that IP address, refer to <b>tf6InterfaceSetSiteId</b> . In the case of an API that returns an address of type <i>sockaddr_in6</i> (i.e. <b>tf6GetLocalIpAddress</b> ), you do not have to set this field, because the API will set it for you. However, in the case of an API that you pass an address of type <i>sockaddr_in6</i> (i.e. <b>tfNgPingOpenStart</b> ), you do need to set this field, if it hasn't already been set by a different API. The API <b>tf6SockaddrSetScopeId</b> can be used to assign the correct value to this field, dependent on which interface you want to use/associate with the address and the scope of the address.

### 6.1.5 sockaddr\_storage

```

struct                sockaddr_storage
{
union {
#ifdef TM_USE_IPV6
    struct sockaddr_in6 ipv6;
#endif /* TM_USE_IPV6 */
#ifdef TM_USE_IPV4
    struct sockaddr_in  ipv4;
#endif /* TM_USE_IPV4 */
}                    addr;
};

#ifdef TM_USE_IPV6
#define ss_len        addr.ipv6.sin6_len
#define ss_family     addr.ipv6.sin6_family
#else /* TM_USE_IPV4 */
#define ss_len        addr.ipv4.sin_len
#define ss_family     addr.ipv4.sin_family
#endif /* TM_USE_IPV4 */
};

```

#### Structure Description

Refer to [RFC2553]. Because the *sockaddr* type isn't big enough to hold an IPv6 address, the application instead uses the *sockaddr\_storage* type to allocate a large enough buffer for an IPv6 address returned by **getsockname()**. Additionally, all of the Treck tFNg "Next Generation" APIs expect IP address parameters to be of type *sockaddr\_storage*, which can hold either an IPv4 or an IPv6 address, the *ss\_family* field of *sockaddr\_storage* being used to differentiate between the two address formats.

#### Structure Members

Member	Description
<i>ss_len</i>	Refer to <i>sockaddr_in6.sin6_len</i> .
<i>ss_family</i>	Refer to <i>sockaddr_in6.sin6_family</i> .
<i>addr.ipv4</i>	IPv4 address specified as type <i>sockaddr_in</i> .
<i>addr.ipv6</i>	IPv6 address specified as type <i>sockaddr_in6</i> .

### 6.1.6 tt6LocalIpAddressCursor

```
typedef struct    ts6LocalIpAddressCursor
{
ttUser32Bit *    opaquePtr1;
ttUser32Bit *    opaquePtr2;
ttUser32Bit      opaqueResultsSet[TM_CURSOR_MAX_ROWS * 4];
ttUser32Bit      opaqueStateInfo;
} tt6LocalIpAddressCursor;
```

#### Structure Description

This structure is used by **tf6GetLocalIpAddress** to keep track of the iteration through the list of multi-homed IPv6 addresses on the interface. This is the type used in the public API. Refer to *tt6ImplLocalIpAddrCursor*, which overlays this structure so that **tf6GetLocalIpAddress** can access internal state information.

#### Structure Members

Member	Description
<i>opaquePtr1</i>	The application doesn't care what this field is used for, and just needs to provide us space for a pointer.
<i>opaquePtr2</i>	The application doesn't care what this field is used for, and just needs to provide us space for a pointer.
<i>opaqueResultsSet</i>	The application doesn't care what this field is used for, and just needs to provide us space for the results set.
<i>opaqueStateInfo</i>	The application doesn't care what this field is used for, and just needs to provide us space for the cursor state information.

## 7 Macros

### 7.1 Performance Macros

Macro Name	Meaning
TM_6_DISABLE_PMTU_DISC	Disable IPv6 Path MTU Discovery.
TM_6_IP_FRAGMENT	Enable IPv6 fragmentation/reassembly code (enabled by default in trsystem.h).
TM_6_USE_DAD	Enables DAD (Duplicate Address Detection) code, requires TM_6_USE_MLD to be enabled.
TM_6_USE_IP_FORWARD	Enable IPv6 forwarding code (enabled by default in trsystem.h)
TM_6_USE_MLD	Enables MLD (Multicast Listener Discovery) code.
TM_6_USE_NUD	Enables NUD (Neighbor Unreachability Detection) code.
TM_6_USE_PREFIX_DISCOVERY	Enables Prefix Discovery, part of stateless address auto-configuration.
TM_OPTIMIZE_MANY_MHOMES	Speed up lookup in the receive-path of IP aliases configured on the interface. This optimizes for situations where the user has configured many (i.e. >50) IP aliases on a single interface. Enabling this functionality causes a small increase in codespace and dataspace usage.
TM_SINGLE_INTERFACE	May be defined when TM_USE_IPV6 is enabled. This performance macro will reduce codesize. However, you will be restricted to using a single interface.
TM_SINGLE_INTERFACE_HOME	This macro is not supported when TM_USE_IPV6 is enabled. Please use TM_SINGLE_INTERFACE instead.
TM_USE_IPV4	Enable IPv4 functionality.
TM_USE_IPV6	Enable IPv6 functionality.
TM_USE_IP_FORWARD	Enable IPv4 forwarding code (enabled by default in trsystem.h)

### 7.2 BSD Socket API Macros

Macro Name	Meaning
AF_INET6/PF_INET6	Refer to [RFC2553]. IPv6 address family/protocol family ID.
IF_NAMESIZE	Refer to [RFC2553]. Max length of interface name. This should correspond to the value of the internal API macro TM_MAX_DEVICE_NAME.
IN6_ARE_ADDR_EQUAL	Refer to [RFC2292]. Macro used to test two IPv6 addresses to see if they are equal.
IN6_IS_ADDR_LINKLOCAL	Refer to [RFC2553]. Macro used to test an IPv6 address to see if it is a link-local scope unicast address.
IN6_IS_ADDR_LOOPBACK	Refer to [RFC2553]. Macro used to test an IPv6 address to see if it is the loopback address.
IN6_IS_ADDR_MC_GLOBAL	Refer to [RFC2553]
IN6_IS_ADDR_MC_LINKLOCAL	Refer to [RFC2553]
IN6_IS_ADDR_MC_NODELOCAL	Refer to [RFC2553]
IN6_IS_ADDR_MC_ORGLOCAL	Refer to [RFC2553]
IN6_IS_ADDR_MC_SITELOCAL	Refer to [RFC2553]

---

IN6_IS_ADDR_MULTICAST	Refer to [RFC2553]. Macro used to test an IPv6 address to see if it is a multicast address.
IN6_IS_ADDR_SITELOCAL	Refer to [RFC2553]. Macro used to test an IPv6 address to see if it is a site-local scope unicast address.
IN6_IS_ADDR_UNSPECIFIED	Refer to [RFC2553]. Macro used to test an IPv6 address to see if it is the unspecified address (i.e. all 0's).
IN6_IS_ADDR_V4COMPAT	Refer to [RFC2553]. Macro used to test an IPv6 address to see if it is an IPv4-compatible IPv6 address.
IN6_IS_ADDR_V4MAPPED	Refer to [RFC2553]. Macro used to test an IPv6 address to see if it represents an IPv4 address (i.e. IPv4-mapped IPv6 address).
IN6ADDR_ANY_INIT	Used at declaration time to initialize an <i>in6_addr</i> structure to the IPv6 wildcard address.
IN6ADDR_LOOPBACK_INIT	Used at declaration time to initialize an <i>in6_addr</i> structure to the IPv6 wildcard address.
INET6_ADDRSTRLEN	Refer to [RFC2553]. Length of IPv6 address string.
INET_ADDRSTRLEN	Refer to [RFC2553]. Length of IPv4 address string.
IPPROTO_AH	Refer to [RFC2292]. IPsec authentication header protocol ID. ([RFC2460].R4:10)
IPPROTO_DSTOPTS	Refer to [RFC2292]. IPv6 Destination options protocol ID. ([RFC2460].R4:10)
IPPROTO_ESP	Refer to [RFC2292]. IPsec encapsulating security payload protocol ID. ([RFC2460].R4:10)
IPPROTO_FRAGMENT	Refer to [RFC2292]. IPv6 fragmentation header protocol ID. ([RFC2460].R4:10)
IPPROTO_HOPOPTS	Refer to [RFC2292]. IPv6 Hop-by-Hop options protocol ID. ([RFC2460].R4:10)
IPPROTO_ICMPV6	Refer to [RFC2292]. ICMPv6 protocol ID. ([RFC2460].R4:10)
IPPROTO_IPV6	Refer to [RFC2292]. IPv6 protocol ID. ([RFC2460].R4:10)
IPPROTO_NONE	Refer to [RFC2292]. IPv6 no next header protocol ID. ([RFC2460].R4:10)
IPPROTO_ROUTING	Refer to [RFC2292]. IPv6 Routing header protocol ID. ([RFC2460].R4:10)
IPV6_JOIN_GROUP	Refer to [RFC2553]. <b>setsockopt</b> IPPROTO_IPV6 <i>protocolLevel</i> option used to join an IPv6 multicast group on a specified interface.
IPV6_LEAVE_GROUP	Refer to [RFC2553]. <b>setsockopt</b> IPPROTO_IPV6 <i>protocolLevel</i> option used to leave an IPv6 multicast group on a specified interface.
IPV6_MULTICAST_HOPS	Refer to [RFC2553]. <b>setsockopt/getsockopt</b> IPPROTO_IPV6 <i>protocolLevel</i> option used to set/get hop the limit for IPv6 multicast packets.
IPV6_MULTICAST_IF	Refer to [RFC2553]. <b>setsockopt/getsockopt</b> IPPROTO_IPV6 <i>protocolLevel</i> option used to set/get the interface to use for outgoing IPv6 multicast packets.
IPV6_UNICAST_HOPS	Refer to [RFC2553]. <b>setsockopt/getsockopt</b> IPPROTO_IPV6 <i>protocolLevel</i> option used to set/get hop the limit for IPv6 unicast packets.
SIN6_LEN	Refer to [RFC2553]. We implement the 4.BSD version of

---

*sockaddr\_in6.***getaddrinfo error codes:**

EAI_ADDRFAMILY	Address family for nodename not supported.
EAI_AGAIN	Temporary failure in name resolution.
EAI_BADFLAGS	Invalid value for ai_flags.
EAI_FAIL	Non-recoverable failure in name resolution.
EAI_FAMILY	ai_family not supported.
EAI_MEMORY	Memory allocation failure.
EAI_NODATA	No address associated with nodename
EAI_NONAME	Nodename nor servname provided, or not known
EAI_SERVICE	servname not supported for ai_socktype
EAI_SOCKTYPE	ai_socktype not supported
EAI_SYSTEM	system error returned in errno
EAI_LAST_ERROR	

**Flags for addrinfo structure**

AI_CANONNAME	Request for canonical name
AI_NUMERICHOST	Don't use name resolution
AI_V4MAPPED	Return IPv4 addresses in IPv4-mapped format if no native IPv6 are found.
AI_ALL	Return all IPv4 and IPv6 addresses found.
AI_ADDRCONFIG	Return IPv4 addresses only if IPv4 is configured on this device; return IPv6 addresses only if IPv6 is configured on this device.

**Flags for getnameinfo**

NI_NUMERICHOST	Don't do a DNS lookup for this address – just return the address as a numeric string.
NI_NAMEREQD	Return an error if the host's name cannot be found.
PF_UNSPEC	Unspecified protocol family
AF_UNSPEC	Unspecified address family
TM_DNS_SERVER	Tertiary DNS server

## 7.3 Socket Extension Macros

Macro Name	Meaning
TM_6_BI_CONF_TUNL_FLAG	Flag used to enable bi-directional configured tunnel functionality, see <b>tf6AddDefaultGatewayTunnel</b> . ([RFC2893].R4:20)
TM_6_DEV_ADDR_CONFIG_COMPLETE	IPv6 address configuration event indicates that the IPv6 address at the specified multi-home index was configured on the interface. Refer to the <b>tfNgConfigInterface</b> parameter <i>dev6AddrNotifyFuncPtr</i> .
TM_6_DEV_ADDR_CONFIG_FAILED	IPv6 address configuration event indicates that the IPv6 address at the specified multi-home index was not configured on the interface because it failed Duplicate Address Detection. Refer to the <b>tfNgConfigInterface</b> parameter <i>dev6AddrNotifyFuncPtr</i> .
TM_6_DEV_ADDR_CONFIG_STARTED	IPv6 address configuration event indicates that configuration of the IPv6 address at the specified multi-home index has been started. Refer to the <b>tfNgConfigInterface</b> parameter <i>dev6AddrNotifyFuncPtr</i> .
TM_6_DEV_ADDR_DEPRECATED	IPv6 address configuration event indicates that an IPv6 address has transitioned state from preferred to deprecated. Refer to the <b>tfNgConfigInterface</b> parameter <i>dev6AddrNotifyFuncPtr</i> .
TM_6_DEV_ADDR_DUP_DETECTED	IPv6 address configuration event indicates that we detected that a different node is using an IPv6 address that we have already configured on the interface. Note that it is up to the user to specify what recovery, if any, is performed when this event occurs. Refer to the <b>tfNgConfigInterface</b> parameter <i>dev6AddrNotifyFuncPtr</i> . ([RFC2462].R5.4.4:10)
TM_6_DEV_ADDR_INVALIDATED	IPv6 address configuration event indicates that an IPv6 address has become invalid, and has been removed from the interface. Refer to the <b>tfNgConfigInterface</b> parameter <i>dev6AddrNotifyFuncPtr</i> .
TM_6_DEV_ADDR_NOTIFY_FUNC_NULL_PTR	( <i>tt6DevAddrNotifyFuncPtr</i> ) 0 Null pointer value for the <i>dev6AddrNotifyFuncPtr</i> parameter of <b>tfNgConfigInterface</b> and <b>tfNgOpenInterface</b> .
TM_6_DEV_CALLED_FROM_APP	Flag passed to <b>tf6Eui64SetInterfaceId</b> or <b>tf6InterfaceSetPhysAddr</b> to indicate that this function is being called from the main loop, i.e. not from the device driver.
TM_6_DEV_CALLED_FROM_DRIVER	Flag passed to <b>tf6Eui64SetInterfaceId</b> or <b>tf6InterfaceSetPhysAddr</b> to indicate that this function is being called from the device driver, typically the device driver open function.
TM_6_DEV_IP_FORW_ENB	Option for <b>tfNgConfigInterface</b> and <b>tfNgOpenInterface</b> . IPv6 device flag (i.e. <i>ttDeviceEntry.dev6Flags</i> ) enables IPv6 forwarding to and from this device.
TM_6_DEV_IP_FORW_MCAST_ENB	Option for <b>tfNgConfigInterface</b> and <b>tfNgOpenInterface</b> . IPv6 device flag (i.e. <i>ttDeviceEntry.dev6Flags</i> ) enables forwarding of IPv6 multicast messages to and from this device. Currently not implemented.

TM_6_DEV_MCAST_HW_LOOPBACK	Option for <b>tfNgConfigInterface</b> and <b>tfNgOpenInterface</b> . IPv6 device flag (i.e. <i>ttDeviceEntry.dev6Flags</i> ) that indicates that the network controller hardware does not support filtering of its own multicasts but instead does loopback of its own multicasts. Generally, you don't want to set this flag, since it causes Duplicate Address Detection to not be as robust. On the other hand, if Duplicate Address Detection always fails regardless of what interface ID you set on the interface or what IPv6 address you try to configure, then specifying this flag will likely fix the problem.
TM_6_DEV_OPTIMIZE_DAD	Option for <b>tfNgConfigInterface</b> and <b>tfNgOpenInterface</b> . IPv6 device flag (i.e. <i>ttDeviceEntry.dev6Flags</i> ) enables optimization of Duplicate Address Detection so that, for IPv6 addresses generated via stateless address auto-configuration, we only perform Duplicate Address Detection on the link-local scope address. ([RFC2462].R5.4:30)
TM_6_USE_AUTO_IID	Option for <b>tfNgConfigInterface</b> and <b>tfNgOpenInterface</b> . IPv6 device flag (i.e. <i>ttDeviceEntry.dev6Flags</i> ) that enables auto-recovery when the IPv6 interface is disabled. Specifically when a Duplicate Address Detection failure occurs on the auto-configured link-local scope address, and this IPv6 device flag was specified, the stack recovers by setting a random interface ID and reopening the interface for IPv6.
TM_6_DEV_OPTIONS_DAD_XMITS	Option for <b>tfInterfaceSetOptions</b> . Controls the number of consecutive Neighbor Solicitation messages sent while performing Duplicate Address Detection on a tentative address. ([RFC2462].R5.1:10)
TM_6_OPTION_ICMP_TX_ERR_LIMIT	Option for <b>tfSetTreckOptions</b> . Rate limit the sending of ICMPv6 error messages (timer-based). ([RFC2463].R2.4:110)
TM_6_OPTION_IP_DEPRECATE_ADDR	Option for <b>tfSetTreckOptions</b> . Enable/disable prevention of any new IPv6 communication from using a deprecated address. ([RFC2462].R5.5.4:50, [RFC2462].R5.5.4:60, [RFC2462].R5.5.4:70)
TM_6_OPTION_IP_FORWARDING	Option for <b>tfSetTreckOptions</b> . A boolean used to enable IPv6 forwarding; indicates whether the IPv6 node is operating as a host or as a router. Note that router functionality is not implemented in phase 1 IPv6.
TM_6_OPTION_IP_FRAGMENT	Option for <b>tfSetTreckOptions</b> . A boolean used to enable IPv6 fragmentation.
TM_6_OPTION_PATH_MTU_TTL	Option to change the time to wait before increasing an IPv6 PMTU estimate.
TM_CURSOR_MAX_ROWS	Maximum number of data rows returned by query using a cursor.
TM_DEV_IP_ASYMMETRICAL	Device flag (i.e. <i>ttDeviceEntry.devFlag</i> ) that indicates that the IP routing table must always be consulted when sending a solicited reply packet (i.e. ICMP Echo Reply) to a unicast destination IP address in response to a request packet (i.e. ICMP Echo Request) received on this device, since the outgoing device that the reply must be sent on may be different from the incoming device that the request was received on.
TM_DNS_TER_SERVER	Tertiary DNS server