Network Kernel Extensions Programming Guide

Drivers, Kernel, & Hardware: User-Space Device Access

2009-08-14

Contents

Figures, Tables, and Listings

Introduction to Network Kernel Extensions Programming Guide

Technology Overview

Network kernel extensions (NKEs) provide a way to extend and modify the networking infrastructure of Mac OS X while the kernel is running, without requiring the kernel to be recompiled, relinked, or rebooted.

NKEs allow you to create modules that can be loaded and unloaded dynamically at specific positions in the network hierarchy. These modules can monitor and modify network traffic, and can receive notification of asynchronous events from the driver layer, such as interface status changes.

This document is primarily of interest to developers who need to extend or modify the Mac OS X networking infrastructure. This includes:

- Adding support for new, non-ethernet interface types.
- Designing custom routing technologies.
- Creating link-layer encryption technologies.

This document assumes a significant understanding of networking concepts, including a basic familiarity with sockets, packet filtering, and so on. It also assumes that you are already familiar with the basics of kernel-level operating systems programming.

Because even minor bugs in kernel-level code can cause serious consequences, including application instability, data corruption, and even kernel panics, the techniques described in this document should be used only if no other mechanism already exists. For example, where possible, IP filtering should generally be done using `ipfw(8)`. Similarly, packet logging should generally be done using `bpf(4)`.

This document is intended to provide supplementary conceptual material specific to network kernel extensions. It is not intended as a reference document, and assumes prior knowledge of Mac OS X kernel extensions (KEXTs). For reference material specific to networking KEXTs, see the document *KPI Reference*. For additional information on Mac OS X KEXTs in general, see the document *Kernel Extension Programming Topics*.

Note: The information provided in this document is *only* relevant for Mac OS X version 10.4 and later. The network kernel extension mechanism used prior to 10.4 is not supported in 10.4 and later. For information on writing network kernel extensions for previous versions of Mac OS X, see *Network Kernel Extensions (legacy)*.

See Also

The following sources provide additional information that may be of interest to developers of network kernel extensions:

- *Kernel Extension Programming Topics*—conceptual information about kernel extensions in Mac OS X.

- *KPI Reference*—reference documentation specific to network kernel extensions and other non-I/O Kit (device driver) KEXTs.
- *Kernel Framework Reference*—reference documentation for I/O Kit device drivers, including network device drivers.
- *The Design and Implementation of the 4.4 BSD Operating System*. M. K. McKusick et al., Addison-Wesley, Reading, 1996.
- *Unix Network Programming,* Second Edition, volume 1. Richard W. Stevens, Prentice Hall, New York, 1998.
- *TCP/IP Illustrated,* volume 1: The Protocols. Richard W. Stevens, Addison-Wesley, Reading, 1994.
- *TCP/IP Illustrated,* volume 2: The Implementation. Richard W. Stevens and Gary R. Wright, Addison-Wesley, Reading, 1995.
- *TCP/IP Illustrated,* volume 3: Other Protocols. Richard W. Stevens, Addison-Wesley, Reading, 1996.

The following websites provide information about the Berkeley Software Distribution (BSD):

- http://www.FreeBSD.org
- http://www.NetBSD.org
- http://www.OpenBSD.org/

Network Kernel Extensions Overview

Network kernel extension (NKE) is a term used to describe any Mac OS X kernel extension that interacts with the networking stack. It is a separately compiled module (produced, for example, by Xcode using the Kernel Extension (KEXT) project type).

Loading a kernel extension is handled by the `kextload(8)` command line utility, which adds the NKE to the running Mac OS X kernel as part of the kernel's address space. Eventually, the system will provide automatic mechanisms for loading extensions. Currently, automatic loading is possible only for I/O Kit KEXTs and other KEXTs that they depend on.

As a kernel extension, an NKE provides initialization and termination routines that the kernel invokes when an NKE is loaded or unloaded. The initialization routine handles initializing local data structures and registering controls, filters, and interfaces. Similarly, the termination routine must free any allocated memory and unregister the extension along with any kernel controls associated with it.

NKE Implementation

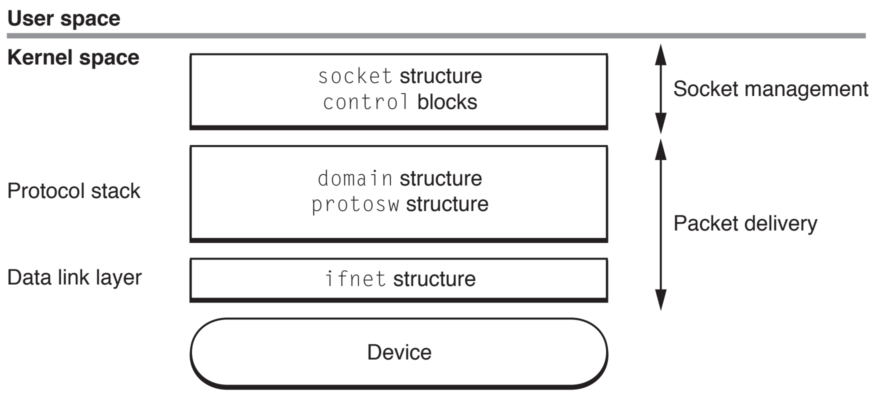
Review of 4.4BSD Network Architecture

Mac OS X is based on the 4.4BSD UNIX operating system. The following structures control the 4.4BSD network architecture:

- `socket` structure, which the kernel uses to keep track of sockets. The `socket` structure is referenced by file descriptors from user mode.
- `domain` structure, which describes protocol families.
- `protosw` structure, which describes protocol handlers. (A protocol handler is the implementation of a particular protocol in a protocol family.)
- `ifnet` structure, which describes a network device and contains pointers to interface device driver routines.

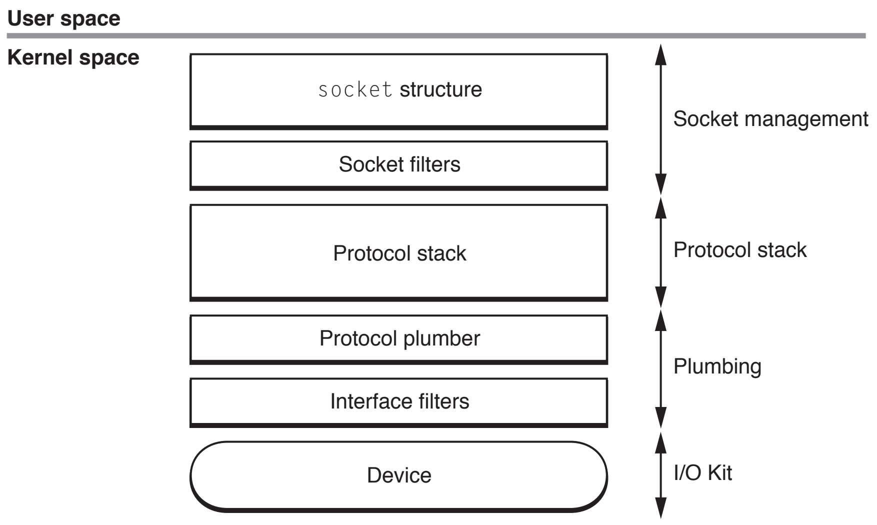
None of these structures is used uniformly throughout the 4.4BSD networking infrastructure. Instead, each structure is used at a specific level, as shown in Figure 1-1 (page 10).

Figure 1-1 Traditional 4.4BSD Networking Architecture



In Mac OS X, the structures themselves are hidden behind opaque types (or in some cases, integer identifiers). However, from a conceptual perspective, equivalent data structures exist and are accessible through accessor functions. The Mac OS X architecture is summarized in Figure 1-2 (page 10).

Figure 1-2 Mac OS X Networking Architecture



The `socket` structure is used to manage the socket. The `domain`, `protosw`, and `ifnet` structures are used to manage packet delivery to and from the network device.

Network KPI Mechanisms

Mac OS X, beginning in version 10.4, supports several networking-related kernel programming interfaces (KPIs). This KPI mechanism consists of opaque data types and functions for manipulating the underlying data structures. Unlike the direct structure access used in previous versions, the KPI mechanism allows for maintaining binary compatibility across OS releases.

Each of the networking KPI mechanisms performs a specific task. The basic networking KPI mechanisms are:

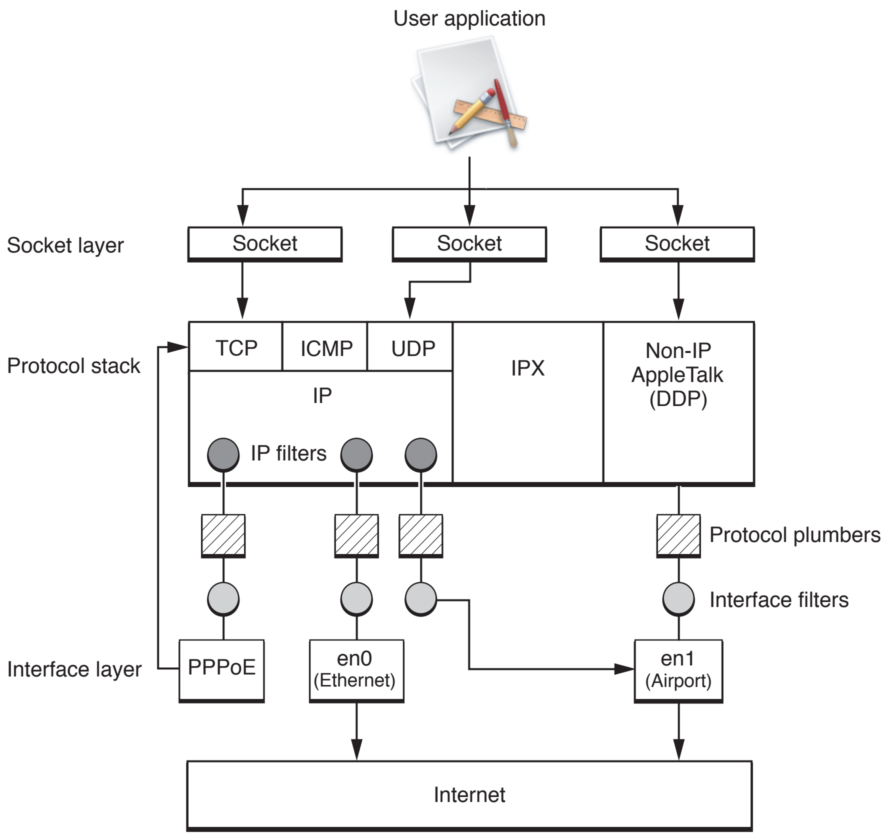
- Socket filter KPI, which permits a KEXT to filter inbound or outbound traffic on a given socket, depending on how they are attached. Socket filters can also filter out-of-band communication such as calls to `setsockopt` or `bind`. The resulting filters lie between the socket layer and the protocol.
- IP filter KPI, which enables a KEXT to perform TCP/IP packet filtering on full (non-fragmented) IP frames. The resulting filters are invoked each time a TCP/IP packet enters the protocol handler layer (usually from the data link layer after the packet is reassembled), and again each time a packet leaves the protocol handler layer (usually going back out to the data link layer).
- Interface filter KPI, which allows a KEXT to add a filter to a specific network interface. These interface filters (previously known as data link NKEs) can passively observe traffic (regardless of packet type) as it flows into and out of the system. They can also modify the traffic (for example, encrypting or performing address translation). They essentially act as filters between a protocol stack and a device.
- Interface KPIs, which allow a KEXT to register a network interface, attach protocols to interfaces, gather information about interfaces, and send packets on an interface. A virtual device created using this mechanism defines a number of media-specific support callbacks (demultiplexing, framing, and pre-output functions such as ARP). Virtual devices can be written entirely using the Interface KPI mechanism.

Note: For hardware devices, you must write an I/O Kit driver (and optionally, an I/O Kit family). For hardware-specific interface types, you should generally add support through subclassing the `IONetworkInterface` and `IONetworkController` classes.

- Protocol plumber KPIs, which allow a KEXT to register code to handle existing protocols on new interface types.

Figure 1-3 (page 12) shows the NKE architecture in detail.

Figure 1-3 NKE architecture



Tracking KEXT Usage

The most important aspect of removing a networking filter, pseudo-interface, and similar components is ensuring that all system resources allocated by the KEXT are returned to the system.

To support the dynamic loading and unloading of KEXTs in Mac OS X, the kernel keeps track of the use of registered filters and similar components by other parts of the system. However, while this protects against dangling dependencies on your KEXT, your KEXT must still release any data structures that it has retained from elsewhere. The KEXT must track its use of resources, such as socket structures and mbufs so that the KEXT's termination routine can eliminate references and return system resources.

Typically, for socket filters, most resources will be specific to a given socket. However, there is a mechanism provided for per-filter allocation. When the networking stack has disposed of all instances of your filter, it will call the filter's `sf_unregistered_func` callback. At that time, your filter should deallocate any resources that are global to the filter.

When the networking stack finishes with a particular instance of your filter, it will call its `sf_detach_func`, `iff_detach_func`, or `ipf_detach_func` callback, respectively. Your extension most not unload until it has received detached or unregistered callbacks for every filter or interface that it has registered. Further, it should track any resources it uses and free those resources before unloading.

Resources that are not per-interface or per-filter can be allocated and freed in the KEXT's start and stop routines.

Instance-Specific Persistent Data

The networking KPI mechanism provides some rudimentary support for tracking memory and data associated with a particular instance of filters. When a filter is attached (regardless of filter type), a cookie is assigned to that particular instance. In the case of socket filters, the attach callback returns this value. For other filter types, the value is passed into the attach function by the caller. While this cookie can contain arbitrary KEXT-specific data, it is generally used to hold a pointer to the data structure of your choice.

The cookie value will be passed as an argument whenever any of your filter's functions are subsequently called on a given filter instance. You can then cast the value to a pointer to the appropriate structure and use this to recover the information stored therein.

As far as the networking stack is concerned, the cookie is a black box that only has meaning within the context of your kernel extension. It will not attempt to manipulate the cookie in any way, and more importantly, if it contains a pointer to a dynamically-allocated object, your KEXT is expected to manage the deallocation of the underlying data object after the filter instance is detached.

KEXT Dependency Information

The dependencies for KPI-based KEXTs are different from those used for pre-KPI KEXTs in prior versions of Mac OS X.

The KPI dependencies for Mac OS X 10.4 are:

Table 1-1 KEXT Dependencies

Bundle Identifier	Version	Description
com.apple.kpi.bsd	8.0.0	BSD APIs
com.apple.kpi.iokit	8.0.0	I/O Kit APIs
com.apple.kpi.libkern	8.0.0	User/Kernel Boundary Crossing APIs
com.apple.kpi.mach	8.0.0	Mach APIs
com.apple.kpi.unsupported	8.0.0	Unsupported and legacy APIs

For dependency versions for other releases of Mac OS X, see *Kernel Extension Programming Topics*.

Memory Buffers, Socket Manipulation, and Socket Input/Output

Most of the networking KPIs enable you to write extensions that change the behavior of the networking stack. The mbuf KPI routines are central to these KPIs, providing functions to manipulate individual network packets within the kernel. You may also sometimes find it useful to perform socket communication in the kernel, for example, to contact a remote server in a network filesystem client such as AFS. The socket KPI routines were designed to help you work with sockets at the kernel level, using mbufs as the fundamental unit of data.

The mbuf KPIs may be unfamiliar territory for you if you are used to user-space network programming. Conceptually, they are just linked lists of objects that can either contain packet data or pointers to external buffers that contain packet data. For incoming traffic, the packet header is also encapsulated in the mbuf structure.

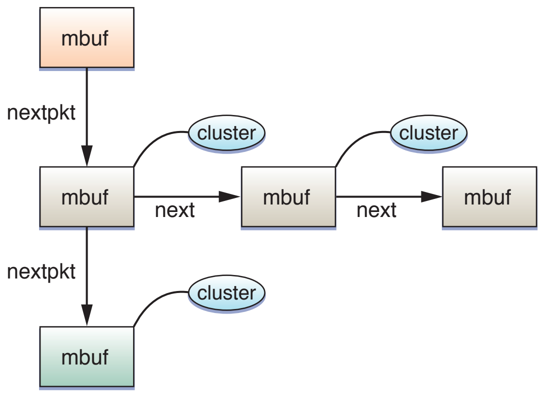
Working with Memory Buffers

All of the networking KPIs are built on top of a shared data structure called a memory buffer, or mbuf. An mbuf is the fundamental unit of data flow through the networking stack and represents a packet (or portion thereof). This section describes the way mbufs and mbuf chains are organized and describes a number of common operations on mbufs. For a complete list of mbuf operations, see `kpi_mbuf.h`.

Structure of an mbuf

A memory buffer, or mbuf, represents the contents of a single data packet. Its structure consists of a packet header (which may be absent for newly-generated outgoing traffic) and a payload (which contains the actual data).

Figure 2-1 A chain of mbuf chains



For smaller payloads, the data may be encapsulated in the mbuf structure itself as an offset from the start of the structure. For a larger payload (beyond the length of the mbuf itself), the payload can be stored separately by associating the mbuf with an external buffer, called a cluster. You can learn whether an mbuf has a cluster by calling `mbuf_flags` and checking to see if the `MBUF_EXT` bit is set.

An mbuf can be part of a singly-linked list, called an mbuf chain. This allows you to chain data together that does not exist in a physically contiguous buffer without copying the data. You can find the next node in an mbuf chain by calling `mbuf_next`. (You cannot obtain the previous node because an mbuf chain is a singly-linked list.)

When moving more data than will fit in a single packet, these mbufs and mbuf chains can, in turn, be combined to form a larger singly-linked list that represents a stream of packets. You can find the next packet by calling `mbuf_nextpkt`.

Note: Not all parts of the networking stack support chains of packets. If you are sending data to an unfamiliar part of the stack, you should err on the side of caution and send packets individually as a single mbuf chain rather than as a chain of mbuf chains.

The packet represented by an mbuf or mbuf chain may be a fragment of a larger packet.

Manipulating an mbuf or mbuf Chain

Using routines in `kpi_mbuf.h`, you can manipulate an mbuf or mbuf chain in a number of ways, including copying data to or from an mbuf or mbuf chain, adding new mbufs to a chain, freeing an mbuf or mbuf chain, shifting data between mbufs in a chain to make a byte range contiguous (if space is available), and so on. This section explains some of the more common mbuf operations.

The most common operation you will need to perform is copying data into and out of an mbuf. To copy data from an mbuf or mbuf chain into a local buffer of your choosing, use `mbuf_copydata`. Be careful not to overflow the buffer. To copy data back into the mbuf or mbuf chain, use `mbuf_copyback`. If needed, the `mbuf_copyback` function will extend the chain to accommodate more data. For an example of how to use `mbuf_copydata`, see Listing 2-1 (page 19).

If you are modifying an existing mbuf in a kernel extension, you should also familiarize yourself with the checksum functions. If your modifications involve changes to packet header information (protocol field, address information, and so on), you must swallow the packet entirely and reinject it.

For payload changes, before modifying the data, your extension should first call `mbuf_inbound_modified`, which disables hardware checksums for a packet, and then call `mbuf_outbound_finalize`, which performs any outstanding operations on the packet so that it is safe for you to modify it. After modifying data, it should call `mbuf_outbound_finalize` again to recompute any checksums invalidated by those data changes.

If you are writing code that must avoid processing an mbuf more than once, the approach you should use depends on the type of filter you are writing. The networking stack will automatically track which IP filters have processed an mbuf. Thus, an IP filter should not see an mbuf more than once. If a socket filter reinjects a packet, however, the system will call all of the socket filters again to process the newly-altered packet.

To guarantee that you will not accidentally process an mbuf more than once, you can use mbuf tagging to attach data to the mbuf as it travels through the system. If you see the mbuf a second time, you can detect that tag and skip that mbuf. This feature is described more fully in "Creating a Socket Filter" (page 32).

Finally, Mac OS X maintains a fair amount of statistical information about these networking-related data structures, including the number of mbufs, the number of clusters, the number of payload bytes available in an mbuf (without using a cluster), and so on. You can obtain this information by calling `mbuf_stats` and examining the information that it returns.

Blocking and Nonblocking mbuf Operations

Many mbuf operations—particularly operations that allocate new buffers—take additional flags of type `mbuf_how_t` to request blocking or nonblocking behavior. If you call them in blocking mode, the operations will block until the requested storage is available. If you call them in nonblocking mode, they will immediately return either the requested storage (if available) or an error code indicating why the operation could not be completed. Here are some tips for choosing whether to request blocking or nonblocking behavior:

- If your code is called from a part of the kernel where blocking is prohibited, you *must* use nonblocking buffer allocation. For example, you must use nonblocking allocation in any function that could be called from the VM system paging path, from an interrupt filter routine, or while holding a spinlock.
- If your code is executing on a performance-critical path, use nonblocking operations:
  - For callback functions marked as "fast path" in the Kernel framework reference documentation.
  - When a higher-level layer can retry the operation without data loss. This will generally result in better performance by allowing other work to occur while you wait.
- If your code is called in a context where failure is not allowed, you should always use blocking operations.
- If none of the above applies, you can choose whichever mode is most compatible with your usage model. When in doubt, choose nonblocking operation.

Working with Sockets in the Kernel

The socket KPIs are very similar to user-space socket functions except that they are prefixed with `sock_`. For example, the KPI function `sock_accept` is nearly identical to the function `accept(2)`. Unlike its user-space equivalent, however, `sock_accept` does not return a socket (file descriptor) through its return value. Instead, it returns a `socket_t` opaque object through a pointer argument and returns an error code (`errno_t`) through its return value. As a result, the KPI functions are more consistent, and your code can perform better error checking and reporting.

Beyond these differences in coding style, kernel-space socket programming requires a great deal more care because of a few subtle differences from their user-space equivalents. These differences, described in the sections that follow, are in two major areas: socket I/O and manipulation of the sockets and file descriptors themselves.

There are also minor differences in the flags that are supported by various functions, as well as other subtle variations in behavior and syntax. These are documented in the API reference for the relevant functions. See *Kernel Framework Reference* for more information.

Manipulating Sockets and File Descriptors

In both user-space code and kernel-space code, a socket is an opaque reference to an underlying object. However, the nature of that reference differs somewhat. In kernel-space code, a socket is represented as an opaque type of type `socket_t`. In user-space code, a socket is represented instead as an integer file descriptor. When user-space code performs any operation on that socket, the kernel looks up that integer in a per-process table, then performs the operation on the underlying kernel-space socket. This relationship between in-kernel sockets and user-space file descriptors affects the way you use sockets in the kernel.

- **In the kernel, you must maintain the relationship between sockets and user-space file descriptors if it exists.** Because you cannot manually tie a socket to a file descriptor in a process, you cannot perform certain operations on user-space sockets from inside the kernel. In particular, if you are manipulating sockets passed in from user-space applications (within a network kernel extension, for example), you cannot safely call `sock_close` on these sockets. If you do call `sock_close`, it leaves a dangling file descriptor, which will probably cause a kernel panic.

  Note: If you were considering calling `sock_close` to redirect a socket to a different location, you can do this by intercepting the connection request with a socket filter and redirecting the stream when the application first opens the socket. Once opened, however, a stream cannot be redirected. For more information, see "Socket filters" (page 31).

- **Kernel-space sockets are not bounded by descriptor limits.** Because kernel sockets are not tied to file descriptors, the number of open sockets inside the kernel is not bounded by limitations on the number of per-process file descriptors.

- **You must close kernel-space sockets to avoid leaks.** Because kernel-space sockets are not tied to a process (and thus are not destroyed when the process exits), the burden of maintaining those sockets falls squarely on the shoulders of the developer of the kernel extension that allocates them.

  If you create a socket with `sock_socket` and do not call `sock_close` on that socket, it will live on *until the next reboot*, stealing precious resources from other kernel extensions and running applications. You *must* clean up after yourself. The kernel cannot do it for you. This means:

  - If you create a socket with `sock_socket`, you must close it with `sock_close`.
  - If you allocate an mbuf, you must either free it explicitly or pass it to a send function that frees it implicitly.

Socket Input and Output

Socket I/O in the kernel differs from user-space socket I/O in two main ways:

- In the kernel, the `read(2)` and `write(2)` system calls are not available. Instead, you must copy data between the `mbuf_t` object and a local buffer using `mbuf_copydata` and `mbuf_copyback`.
- In the kernel, asynchronous socket reads are handled differently. The kernel does not provide the equivalent of `select(2)` for writing wait loops. Instead, it provides a callback mechanism that calls a function in your extension whenever data becomes available on a socket.

If your code is in a performance-critical part of the kernel (as opposed to a call from user space), you should generally perform socket I/O asynchronously. In the kernel, this asynchronous I/O is based on a callback mechanism, using callbacks of the type `sock_upcall`. Listing 2-1 shows how to open a socket asynchronously and perform an asynchronous read on the socket using `sock_receivembuf`.

Listing 2-1 Reading from a kernel-space socket asynchronously

```
#include <kern/debug.h>
#include <sys/errno.h>
#include <sys/kpi_mbuf.h>
#include <sys/kpi_socket.h>
#include <net/kpi_protocol.h>
#include <net/ethernet.h>
#include <sys/param.h>
#include <sys/filio.h>

#define ULTIMATE_ANSWER 0x00000042

/* Forward declarations */
errno_t set_nonblocking(socket_t so, int value);
static void my_sock_callback(socket_t so, void* cookie, int waitf);

/* This function opens a connection and sets it up to wait
   for data.  When data is received, the network stack will
   call the callback function my_sock_callback(). */
errno_t open_socket_and_start_listener(void)
{
    socket_t so;
    errno_t err;
    int protocol = 0; // usually the right choice
    struct sockaddr to;
    uint32_t cookie = ULTIMATE_ANSWER; // Normally, we would
                                       // point to a private
                                       // data structure here.

    if ((err = sock_socket(PF_INET, SOCK_STREAM, protocol,
        (sock_upcall)&my_sock_callback, (void *)cookie, &so))) {
            return err;
    }

    /* Set the socket to non-blocking mode */
    set_nonblocking(so, 1);

    /* ... Fill in sockaddr value here ... */

    err = sock_connect(so, &to, MSG_DONTWAIT);
    if (err == EINPROGRESS) return 0; // it worked.
    return err;
}

/* This function is called when data is available on the socket.
   It reads data from the socket. */
static void my_sock_callback(socket_t so, void* cookie, int waitf)
{
    errno_t err;
    size_t len = sizeof(value);
    mbuf_t data;
    int value;
    /* The socket should have some data available now. */
    if (cookie == (void *)ULTIMATE_ANSWER) {
        // This socket's cookie matches the desired magic value,
        // so read data from the socket here.
        err = sock_receivembuf(so, NULL, &data, MSG_WAITALL, &len);
        if (err == EWOULDBLOCK) {
            // The kernel hasn't seen enough data yet.
            return;
        } else if (err || len < sizeof(value)) {
            /* This example does no error recovery.  Your code should. */
            panic("Something is very wrong.  Maybe the other end closed the
connection....\n");
        }
    }

    // Copy the data from the mbuf chain into local storage.
    err = mbuf_copydata(data, 0, sizeof(value), &value); // Copy 4 bytes at
start

    // Call a function with the value received.
    dont_panic(htonl(value));

    // We no longer need this socket, so close it.
    sock_close(so);
}

/* This is a short example of how to use the sock_ioctl()
   function on a socket within the kernel.  This is the
   KPI equivalent of the ioctl() system call. */
errno_t set_nonblocking(socket_t so, int value)
{
    errno_t err;
    int val = value; // taking the address of parameters is bad

    if (value != 0 && value != 1) return EINVAL;
    err = sock_ioctl(so, FIONBIO, &val);
    return err;
}
```

Listing 2-1 also shows the use of the mbuf routine `mbuf_copydata`. This routine copies data from an mbuf chain to a buffer. When using this function, you *must* make sure you do not overflow the destination buffer.

Handling incoming connections works similarly. If you want to block until a connection is pending, you can use `sock_accept` in blocking mode to wait for an incoming connection. If you would prefer to handle incoming connections asynchronously, you can pass in a callback of type `sock_upcall` when you create the socket with `sock_socket`. That callback will be called whenever a client connects to your listen socket. From your callback handler, you should then call `sock_accept` (optionally with the `MSG_DONTWAIT` flag).

Writing data to a socket is similar to reading data synchronously. The steps for writing data to a socket are as follows:

1. If you do not already have an mbuf, call `mbuf_allocpacket` to create an mbuf with a cluster to hold external data. This need not be as large as your data; the next step will expand the chain as needed.

2. Call `mbuf_copyback` to copy data from a local buffer into the mbuf. To copy data from additional buffers, simply repeat this step for each subsequent buffer, specifying the offset from the start of the mbuf where the contents of the buffer should be stored.

3. Call `sock_sendmbuf` to send the data.

For More Information

For information about user-space socket programming in general, you should read the UNIX Socket FAQ. This bulletin board provides the answers to a lot of basic, intermediate, and advanced user-space socket programming questions, and includes numerous examples. You can also find details about user-space socket functions in *Mac OS X Man Pages*.

To learn more about the KPI networking functions, read *Kernel Framework Reference*—in particular, the kernel mbuf data structures and associated functions, described in `kpi_mbuf.h`, and socket APIs, described in `kpi_socket.h`.

KEXT Controls and Notifications

This chapter describes two mechanisms for interacting with a network kernel extension: the kernel control and kernel event APIs. These socket-based APIs allow you to communicate with a KEXT and receive broadcast notifications from the KEXT, respectively.

To support this communication, Mac OS X defines a new socket domain—the `PF_SYSTEM` domain—to provide a way for applications to configure and control KEXTs. The `PF_SYSTEM` domain, in turn, supports two protocols, `SYSPROTO_CONTROL` and `SYSPROTO_EVENT`.

The kernel control (`kern_control`) API, which uses the `SYSPROTO_CONTROL` protocol, allows applications to configure and control a KEXT.

The kernel event (`kern_event`) API, which uses the `SYSPROTO_EVENT` protocol, allows applications and other KEXTs to be notified when certain kernel events occur. It should be used when multiple clients need to know about a given event, and is not intended as a point-to-point communication mechanism. In general, the kernel control API is preferred, as it provides bidirectional communication.

For detailed reference documentation on these APIs, see *Kernel Framework Reference*.

Using the Kernel Control API for KEXT Control

The kernel control API is a bidirectional communication mechanism between a user space application and a KEXT. This section describes this API at the kernel level and the user space level.

Supporting Kernel Controls in Your KEXT

Supporting kernel controls in a KEXT is relatively straightforward.

In the KEXT's start function, you must register a kernel control structure using the `ctl_register` function. The `ctl_register` function is defined in `<sys/kern_control.h>` as follows:

```
int ctl_register(struct kern_ctl_reg *userctl,
            kern_ctl_ref *ctlref);
```

The `kern_ctl_reg` structure contains three fields that are used to identify the control. The fields `ctl_id` and `ctl_name` can be shared across multiple controls.

The final field, `ctl_unit`, contains a value that is specific to a given control. A control can be registered multiple times with the same `ctl_id`, but for each instance a different unit number must be used. For dynamically-allocated control IDs, this value is filled in automatically.

Other fields of the `kern_ctl_reg` structure contain handler functions that you must create to handle various control requests.

The structure's fields are defined as follows:

`ctl_name`
: a bundle ID string for your control of up to `MAX_KCTL_NAME` bytes (including the terminating null). This may be used to generate `ctl_id`.

`ctl_id`
: a unique 4 byte ID for the control. (See note below.)

`ctl_unit`
: the unit number for the control. The value is automatically assigned for dynamically-allocated `ctl_id` values.

`ctl_flags`
: flags that affect the behavior of a control. You can set the `CTL_FLAG_PRIVILEGED` flag to require that the user have admin privileges to contact the control.

  For more TCP-like behavior, the flag `CTL_FLAG_REG_SOCK_STREAM` may be specified to indicate that the control should be registered for stream connections rather than datagrams. Note, however, that if you set `CTL_FLAG_REG_SOCK_STREAM`, you *must* connect to the control using `SOCK_STREAM` instead of `SOCK_DGRAM`.

`ctl_sendsize`
: size of buffer reserved for sending messages. A value of 0 indicates that the default size should be used.

`ctl_recvsize`
: size of buffer reserved for receiving messages. A value of 0 indicates that the default size should be used.

`ctl_connect`
: called when the client process calls connect on the socket with the ID/unit number of the registered control.

`ctl_disconnect`
: called when the user client process closes the control socket.

`ctl_send`
: called when the user client process writes data to the socket.

`ctl_setopt`
: called when the user client process calls `setsockopt` to set the control configuration.

`ctl_getopt`
: called when the user client process calls `getsockopt` on the socket.

Note: You may use either a registered Creator ID (available from the Apple Developer Creator ID web page at http://developer.apple.com/dev/cftype/) or you may use a dynamically-assigned ID.

It is strongly recommended that you use a dynamically-assigned ID. This is the default behavior. In that case, the memory referenced by the `ctl_id` field will be overwritten with the dynamically-generated ID value when `ctl_register` returns.

If you need to use a registered ID, you must set the `CTL_FLAG_REG_ID_UNIT` flag in `ctl_flags`. If this flag is set, the value of `ctl_name` will be ignored.

On successful return, the second parameter, `ctlref`, will contain a reference to the registered kernel control. This reference must be used to unregister the control, and is also passed as an argument to any callbacks when they are called.

It is possible to take advantage of kernel control naming to allow processes to interact with a KEXT in different ways. A KEXT may, for example, register a root-only control for configuring the KEXT. It might register a second control, available to any process, for gathering statistics. Each instance of the control will have a different `ctlref`, and this value can then be used to determine which behavior to use.

When the kernel control receives a connection from a user-space process, the control's `ctl_connect_func` callback is called. In this function, you should determine the unit number associated with the connection so that you can later send data back to the connecting process. You should then create a data structure (of your choosing) to store connection-specific data, and should return this structure by assignment through the `void **` handle passed in as the third parameter. This value will be passed to the other callbacks when they are called.

At this point, the user process can communicate with the control using `getsockopt(2)`, `setsockopt(2)`, `read(2)`/`recv(2)`, and `write(2)`/`send(2)` on the socket. With the exception of `recv(2)` (which reads data from a queue), calls in user space to these functions result in a kernel-space call to the equivalent callbacks in the control, `ctl_getopt_func`, `ctl_setopt_func`, and `ctl_send`, respectively.

The kernel process can, in turn, call a number of functions to send data back to the user space process. This data can be read by the user process using the `read(2)` or `recv(2)` system calls. In particular, you can use `ctl_enqueuedata` and `ctl_enqueuembuf` to queue up data to send to the user space process, and `ctl_getenqueuespace` to find out how much free space is available in the queue.

When the user process closes the communication socket to the control, the `ctl_disconnect_func` callback is called. At this point, the control should free any connection-specific resources that it has allocated.

Listing 3-1 (page 25) shows some basic example functions to use as a starting point:

Listing 3-1 A basic `kern_control` example

```
errno_t error;
struct kern_ctl_reg     ep_ctl; // Initialize control
kern_ctl_ref     kctlref;
bzero(&ep_ctl, sizeof(ep_ctl));  // sets ctl_unit to 0
ep_ctl.ctl_id = 0; /* OLD STYLE: ep_ctl.ctl_id = kEPCommID; */
ep_ctl.ctl_unit = 0;
strcpy(ep_ctl.ctl_name, "org.mklinux.nke.foo");
ep_ctl.ctl_flags = CTL_FLAG_PRIVILEGED & CTL_FLAG_REG_ID_UNIT;
ep_ctl.ctl_send = EPHandleWrite;
ep_ctl.ctl_getopt = EPHandleGet;
ep_ctl.ctl_setopt = EPHandleSet;
ep_ctl.ctl_connect = EPHandleConnect;
ep_ctl.ctl_disconnect = EPHandleDisconnect;
error = ctl_register(&ep_ctl, &kctlref);

/* A simple setsockopt handler */
errno_t EPHandleSet( kern_ctl_ref ctlref, unsigned int unit, void *userdata, int opt,
void *data, size_t len )
{
    int    error = EINVAL;
#if DO_LOG
    log(LOG_ERR, "EPHandleSet opt is %d\n", opt);
#endif

    switch ( opt )
    {
        case kEPCommand1:               // program defined symbol
```

```
            error = Do_First_Thing();
            break;

        case kEPCommand2:               // program defined symbol
            error = Do_Command2();
            break;
    }
    return error;
}

/* A simple A simple getsockopt handler */
errno_t EPHandleGet(kern_ctl_ref ctlref, unsigned int unit, void *userdata, int opt,
void *data, size_t *len)
{
    int    error = EINVAL;
#if DO_LOG
    log(LOG_ERR, "EPHandleGet opt is %d *****************\n", opt);
#endif
    return error;
}

/* A minimalist connect handler */
errno_t
EPHandleConnect(kern_ctl_ref ctlref, struct sockaddr_ctl *sac, void **unitinfo)
{
#if DO_LOG
    log(LOG_ERR, "EPHandleConnect called\n");
#endif
    return (0);
}

/* A minimalist disconnect handler */
errno_t
EPHandleDisconnect(kern_ctl_ref ctlref, unsigned int unit, void *unitinfo)
{
#if DO_LOG
    log(LOG_ERR, "EPHandleDisconnect called\n");
#endif
    return;
}

/* A minimalist write handler */
errno_t EPHandleWrite(kern_ctl_ref ctlref, unsigned int unit, void *userdata, mbuf_t m,
 int flags)
{
#if DO_LOG
    log(LOG_ERR, "EPHandleWrite called\n");
#endif
    return (0);
}
```

Connection from the Client Process

Adding `kern_control` support in your NKE is only half of the story. The other half is actually using this support from a client application.

To communicate with an NKE, you must first open a `PF_SYSTEM` socket using the `socket` call as follows:

```
fd = socket(PF_SYSTEM, SOCK_DGRAM, SYSPROTO_CONTROL);
```

Note: A kernel control may register either a datagram or stream control (`SOCK_DGRAM` or `SOCK_STREAM`). If you registered your control with `CTL_FLAG_REG_SOCK_STREAM`, you should specify `SOCK_STREAM` here. Otherwise, specify `SOCK_DGRAM`.

Next, your application must associate the socket with a particular kernel control. To do this, the client process should call `connect(2)` with the file descriptor returned from the `socket(2)` call, along with a filled in `sockaddr_ctl` structure containing the ID and unit number of the NKE's kernel control.

For example:

```
sockaddr_ctl addr;

/* (initialize addr here) */

result = connect(fd, (struct sockaddr *)&addr, sizeof(addr));
```

The second parameter, of type `sockaddr_ctl`, should be filled in as follows:

```
addr.sc_len = sizeof(struct sockaddr_ctl);
addr.sc_family = AF_SYSTEM;
addr.ss_sysaddr = AF_SYS_CONTROL;
addr.sc_id = MY_ID;     // set to value of ctl_id registered by the NKE in
                        // the ctl_register call described above.
addr.sc_unit = MY_UNIT; // set to the unit number registered by the NKE
                        // in the ctl_register call described above.
```

Of course, in the case of a dynamically-generated control ID, you must obtain the value for `sc_id` using the `CTLIOCGINFO` ioctl, as shown in Listing 3-1 (page 25). When using a dynamically-generated control ID, the unit number is ignored. The stack will automatically pick an unused unit number and fill in the `sc_unit` field before passing the `connect(2)` call to the kernel control's connect callback. While the kernel side must keep track of the unit number for sending data back to the client, from the client's perspective, the unit number is unused.

Now that a communication channel is in place, the client process may use the `setsockopt` call to send commands to the NKE, or the `getsockopt` call to obtain status information from the NKE. The NKE defines which socket option names it will handle. The client process should pass only supported option names to the NKE in the `setsockopt` call. However, for safety, it is the responsibility of the NKE to ignore options that it does not understand, returning `EOPNOTSUPP`.

Listing 3-2 (page 27) shows a code example for opening a `PF_SYSTEM` socket to communicate with an NKE.

Listing 3-2 Opening a `PF_SYSTEM` socket to use with `kern_control`

```
struct sockaddr_ctl       addr;
int                       ret = 1;

fd = socket(PF_SYSTEM, SOCK_DGRAM, SYSPROTO_CONTROL);
if (fd != -1) {
    bzero(&addr, sizeof(addr)); // sets the sc_unit field to 0
    addr.sc_len = sizeof(addr);
    addr.sc_family = AF_SYSTEM;
```

```
        addr.ss_sysaddr = AF_SYS_CONTROL;
#ifdef STATIC_ID
        addr.sc_id = kEPCommID;  // should be unique - use a registered Creator ID here
        addr.sc_unit = kEPCommUnit;  // should be unique.
#else
        {
            struct ctl_info info;
            memset(&info, 0, sizeof(info));
            strncpy(info.ctl_name, MYCONTROLNAME, sizeof(info.ctl_name));
            if (ioctl(fd, CTLIOCGINFO, &info)) {
                perror("Could not get ID for kernel control.\n");
                exit(-1);
            }
            addr.sc_id = info.ctl_id;
            addr.sc_unit = 0;
        }
#endif

        result = connect(fd, (struct sockaddr *)&addr, sizeof(addr));
        if (result) {
           fprintf(stderr, "connect failed %d\n", result);
        }
    } else { /* no fd */
            fprintf(stderr, "failed to open socket\n");
    }

    if (!result) {
        result = setsockopt( fd, SYSPROTO_CONTROL, kEPCommand1, NULL, 0);
        if (result){
            fprintf(stderr, "setsockopt failed on kEPCommand1 call - result was %d\n",
 result);
        }
    }
```

Using the kern_event API for Kernel Notifications

The kernel event notification mechanism, or `kern_event`, is a lightweight mechanism that allows applications to be notified when certain kernel events occur. It is a one-shot event from kernel space to user space that is broadcast to all processes that are listening. For bidirectional communication, you must use the `kern_control` API, described in "Using the Kernel Control API for KEXT Control" (page 23).

This API is relatively straightforward. At initialization time, your NKE should call `kev_vendor_code_find` with the bundle name of your NKE (up to 200 characters in length). It will return a unique identifier that your KEXT should use to identify any notifications that it posts. This identifier value is not persistent across reboots.

Once you have a vendor code, your NKE can post notifications. To post a notification, your NKE calls `kev_message_post` with a `kev_msg` structure containing the vendor code obtained previously, along with the event's class, subclass, event code, and up to five pieces of data of arbitrary length associated with the event.

You can define your own class and subclass values as appropriate for your NKE. The Apple-defined class values used by kernel events built into Mac OS X can be found in the header file `kern_event.h`.

Note: Kernel extensions cannot post `KEV_VENDOR_APPLE` events.

Receiving Kernel Event Notifications

To receive kernel notifications in a client application, you must first create a kernel event socket as follows:

```
fd = socket(PF_SYSTEM, SOCK_RAW, SYSPROTO_EVENT);
```

Note: Kernel event notifications can only be received by processes running as the root user.

Once you have created this socket, you can use this to receive event notifications. There are several ioctls available to help you filter notifications:

- `SIOCGKEVFILT`—get the kernel event filter for this socket.
- `SIOCGKEVID`—get the current event ID pending on the socket. Each event will have a different ID.
- `SIOCGKEVVENDOR`—look up a vendor code.
- `SIOCSKEVFILT`—set the kernel event filter for this socket.

For example, to set the event filter to filter only for Apple-generated events from AppleTalk, you might do the following:

```
struct kev_request req;
req.vendor_code=KEV_VENDOR_APPLE;
req.kev_class=KEV_APPLESHARE_CLASS;
req.kev_subclass=KEV_ANY_SUBCLASS;

if (ioctl(fd, SIOCSKEVFILT, &req)) {
    perror("SIOCSKEVFILT");
    exit(-1);
}
```

Using the `SIOCGKEVFILT` ioctl is similar:

```
struct kev_request req;

if (ioctl(fd, SIOCGKEVFILT, &req)) {
    perror("SIOCSKEVFILT");
    exit(-1);
}
printf("The current filter is vendor code %d, class %d, subclass %d\n",
    req.vendor_code, req.class, req.subclass);
```

To look up a vendor code for another vendor, you might do the following:

```
struct kev_vendor_code vc;
strcpy(vc.vendor_string, "org.mklinux.driver.swim3");
if (ioctl(fd, SIOCGKEVVENDOR, &vc)) exit(-1);
printf("Vendor code returned was %d\n", vc.vendor_code);
```

Finally, to obtain the next event ID from the socket, you might do something like this:

```
uint32_t id;
if (ioctl(fd, SIOCGKEVID, &id)) exit(-1);
printf("ID returned was %d\n", id);
```

Implementing a Preference File for an NKE

Developers often ask how an NKE can open a "preference file" in an NKE's start function. Under the existing architecture, the NKE cannot reliably access a preference file. When the system starts the NKE, there are no APIs that the NKE can use to open a file and read preference information.

The proper way to dynamically configure an NKE is with a startup daemon or other application-level process. The daemon finds the NKE using the kernel control (`kern_control`) mechanism described in "Using the Kernel Control API for KEXT Control" (page 23), and passes in configuration information that the NKE may require.

Helpful Tips

To avoid crashes, unexplained behavior, and other pitfalls, there are a few simple rules you should follow when using `kern_control` and `kern_event` in your NKE.

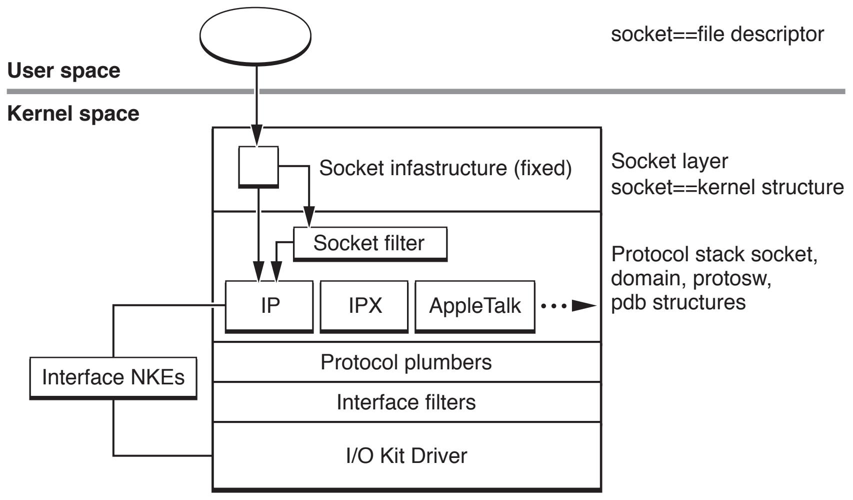
Unregister your control.
: When someone tries to talk to you after your KEXT is unloaded, a kernel panic ensues. You must use `ctl_deregister` to unregister your control before your NKE is unloaded. This call will fail if there are clients still connected to your kernel control.

The maximum data size for events is 2KB.
: Data passed with the `kern_event` APIs must be sent in chunks no larger than the `mbuf` cluster size, or 2KB. Otherwise, truncation will occur.

Socket Filters

A socket filter is a filter associated with a particular socket, as shown in Figure 4-1 (page 31). These extensions can filter inbound or outbound traffic on a socket. They also can filter out-of-band communication, including calls to `setsockopt(2)`, `getsockopt(2)`, `ioctl(2)`, `connect(2)`, `listen(2)`, and `bind(2)`.

Figure 4-1 Socket filters in the Networking Stack



Socket filters can operate in one of two modes: programmatic or global. A global filter is automatically enabled for new sockets of the type specified for the filter. A programmatic filter is enabled only under program control by using `setsockopt(2)` on a specific socket. (Within the code itself, the only difference between global and programmatic filters is whether the flag `SFLT_GLOBAL` or `SFLT_PROG` was set in the filter's `sf_flags` field.)

When a KEXT calls `sock_socket` or an application calls `socket(2)` to create a socket, any global filters associated with the corresponding protocol are attached to the socket structure. Depending on whether the filter is filtering incoming or outgoing data, it will alter the data either just before the incoming data is stored into the socket's buffer or just after outgoing data is retrieved from that buffer by the kernel.

Alternately, an application can call `setsockopt(2)` using socket option `SO_NKE` to insert a programmatic filter into that socket's filter chain, as follows:

```
setsockopt(s, SOL_SOCKET, SO_NKE, &so_nke, sizeof (struct so_nke);
```

The `so_nke` structure is defined as follows:

```
struct so_nke {
    unsigned int nke_handle;
    unsigned int nke_where;
    int nke_flags;
};
```

The values of nke_where and nke_flags are ignored. These fields are maintained only for compatibility.

The nke_handle specifies the filter to be linked to the socket. It is the programmer's task to locate the KEXT containing the appropriate filter and make sure that it is loaded.

The nke_handle values are assigned by Apple Computer from the same name space as the type and creator codes used in Mac OS 8 and Mac OS 9 and using the same registration mechanism.

However, you can also use the kernel event ID allocation mechanism to get a unique handle value for a socket filter. A user-space application can then use the SIOCGKEVVENDOR ioctl on a kernel event socket to determine the dynamic handle value for a given socket filter. This mechanism is described in "Using the kern_event API for Kernel Notifications" (page 28).

Creating a Socket Filter

The life cycle of a socket filter can be summed up as follows:

- Socket filters are installed in the kernel by calling sflt_register, typically from the filter's initialization routine.
- Later, when the filter is instantiated on a socket, the protocol calls the filter's sf_attach_func callback. This callback may return a unique cookie through its first parameter that can be used for tracking storage specific to a given filter instance (attached to a specific socket).
- When the filter is detached (whether through the filter being unregistered, the socket being closed, or the filter being explicitly detached from the socket), the filter's sf_detach_func callback is called. At this point, the filter should free any socket-specific resources that it has allocated (generally in sf_attach_func).
- The socket filter may, at some point, decide that it wishes to be unloaded. If so, it should call sflt_unregister. This will prevent the filter from being attached to new sockets in the future and will begin the process of detaching the filter from existing sockets.

As part of the call to sflt_register, your KEXT passes in a struct sflt_filter object. This structure contains a number of fields that hold various callbacks and flags related to your filter.

Note: Socket filters support both global and programmatic modes. To register a programmatic socket filter, set the flag SFLT_PROG in the sf_flags field of the filter declaration structure. To register a global socket filter, set the flag SFLT_GLOBAL.

Each socket filter contains a number of callbacks (function pointers). These callbacks are called automatically when the corresponding socket functions are called. The callbacks permit the filter to selectively intercept socket operations.

For example, the prototype for sf_bind_func looks like this:

```
int (*sf_bind_func)(void *cookie, socket_t so, const struct sockaddr *to);
```

The kernel's sobind function calls the filter's sf_bind_func callback with the cookie value that the filter's sf_attach_func callback returned when the filter was first attached, along with a socket instance (so) and the name of the local endpoint being bound (to).

Most of these callbacks can return an integer value (with the exception of `detach` and `notify`, which are assumed to always succeed). A return value of zero is interpreted to mean that the caller should continue processing as usual. A non-zero return value is interpreted as an error (as defined in `<sys/errno.h>`) that causes the processing of the packet or socket operation to halt; the error then propagates up through the stack.

The one exception is the return value `EJUSTRETURN`. If you return this value, the calling function (for example, `sobind`) returns at that point with a value of zero (no error). In this way, a filter can "swallow" a packet or an operation. A filter may reinject the data or operation at a later time. For other non-zero return values, the calling function returns the non-zero error code.

When any filter swallows and reinjects a packet or operation, it should expect the relevant filter function to be called again on the injected data or operation. This may occur multiple times—each time the packet is swallowed and reinjected.

Many filters (encapsulation, for example) naturally lend themselves to detecting reinjected packets. In other situations, you can use the mbuf tag functionality to make it easy to spot reinjected traffic.

Note: You should not confuse mbuf tags with virtual LAN (VLAN) tags. Both mechanisms allow you to associate data with an mbuf, but they serve very different purposes. Virtual LAN tagging is intended to be used by hardware interface drivers to identify packets as originating from a virtual LAN. If you care about the source of such a packet, you may need to read this value. In general, this value should only be set from within a network driver. You should never overload this to hold non-VLAN information.

To use mbuf tagging, you must first set a tag identifier for your KEXT in its start routine using the `mbuf_tag_id_find` function Then, at the entry to your `sf_data_in_func` callback, use the `mbuf_tag_find` function to see if your filter has already tagged this packet. If not, it should process the packet. Otherwise, your filter function should return 0 immediately.

Once you have finished processing the packet, you should call `mbuf_tag_allocate` on the packet header mbuf to tag the packet, indicating that you have already processed it. When the mbuf is later freed, any tag references will also be deallocated.

Important: If your KEXT swallows and reinjects packets, it *must* reinject those packets in the order that they arrived on the socket.

The `tcplognke` sample provides an example of how to properly swallow and reinject packets.

Socket Filter Example: tcplognke

The `tcplognke` filter is a socket filter which is invoked for each TCP socket. It records detailed information about each connection, including the number of bytes sent to and from the system, the time the connection was up, and the remote IP address.

The tcplog utility demonstrates the use of the `PF_SYSTEM` socket to enable/disable logging in the `tcplognke`, to read log information from the filter, and to specify different logging criteria.

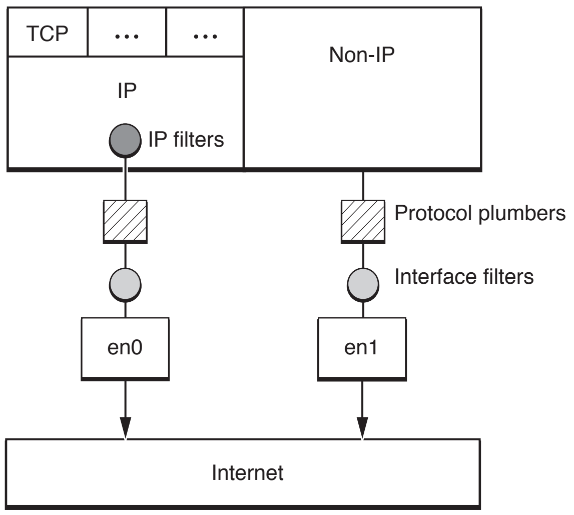
When `tcplognke` is loaded and initialized, it installs itself as a global filter for the TCP protocol and registers a kernel control. The `tcplognke` filter then keeps a buffer of connection records. If no control program attaches to it, the buffer is continually overwritten as connections are established and terminated. To retain or view the information that the `tcplognke` filter gathers, use the enclosed `tcplog` command line utility. The tool configures the `tcplognke` filter to send log records to the `tcplog` program. The `tcplog` tool then loops, displaying and writing log records as the `tcplognke` filter creates them.

The source code for the `tcplognke` filter and for the `tcplog` command-line utility are available from the ADC sample code website. See the Read Me file with the *tcplognke* sample code for more instructions on the design and use of the sample KEXT.

IP Filters

An IP filter is used to filter inbound or outbound IP traffic. It resides within the IP protocol stack, as shown in Figure 5-1 (page 35). For inbound traffic, it is called after an IP packet has been reassembled. For outbound traffic, it is called just prior to IP fragmentation. If IPSec processing is required for a given packet, the filter is called twice—immediately before and after any IPSec processing.

Figure 5-1 IP Filters in the Networking Stack



The Anatomy of an IP Filter

There are two basic categories of IP filters: IPv4 filters and IPv6 filters. With the exception of their handling of addresses, they are essentially equivalent. The same basic data structure, `ipf_filter`, is used to describe both.

The data structure contains five fields: `cookie`, `name`, `ipf_input`, `ipf_output`, and `ipf_detach`.

The first field, `cookie`, can contain arbitrary data. Your KEXT assigns it a value when it attaches the filter to the IP stack. The IP stack then passes that value as an argument whenever the networking stack calls any function in your KEXT. This allows a single filter to have multiple behaviors depending on where it is attached by testing values stored in the cookie.

The structure referenced by this field can be arbitrarily defined by your KEXT. As far as the kernel is concerned, it is essentially a void pointer. This mechanism is commonly used to store information about memory allocations associated with a particular filter instance.

The second field, `name`, is the name of your filter. This is used only for debugging purposes, but should always be filled in. It should contain either the identifier for the KEXT or something similar, for ease of identification.

The remaining fields, `ipf_input`, `ipf_output`, and `ipf_detach`, are pointers to callback functions in your KEXT. Those callbacks are called whenever your filter is asked to handle inbound packets, handle outbound packets, or detach, respectively.

The `ipf_input`, `ipf_output`, and `ipf_detach` function pointers are described in their data type declarations—respectively, `ipf_input_func`, `ipf_output_func`, and `ipf_detach_func`.

Generally, your `ipf_input_func` callback will be called as soon as a packet has been identified as being a IP packet and reassembled. Similarly, your `ipf_output_func` function will be called just prior to sending it to the data link interface layer (where it may be further processed by interface filters). However, in some cases, such as IPSec encapsulation, your IP filter will be called once as each layer of encapsulation is decoded.

A registered filter is identified by the opaque type `ipfilter_t`. This is used later when you unregister the filter.

IP Filter Gotchas

There are several quirks specific to modifying traffic in an IP filter. Some of these include:

Reinjecting modified traffic

If your filter modifies the protocol of inbound traffic or the destination of outbound traffic, the packet may be misdelivered as a result of caching in the IP stack.

To prevent this problem, your filter must use `ipf_inject_input` or `ipf_inject_output`, as appropriate. Your `ipf_input_func` or `ipf_output_func` callback should then swallow the previous version by returning `EJUSTRETURN`.

Packet Fragmentation

IP filters only receive reassembled packets. It is not possible to filter on packet fragments.

Filter Loops

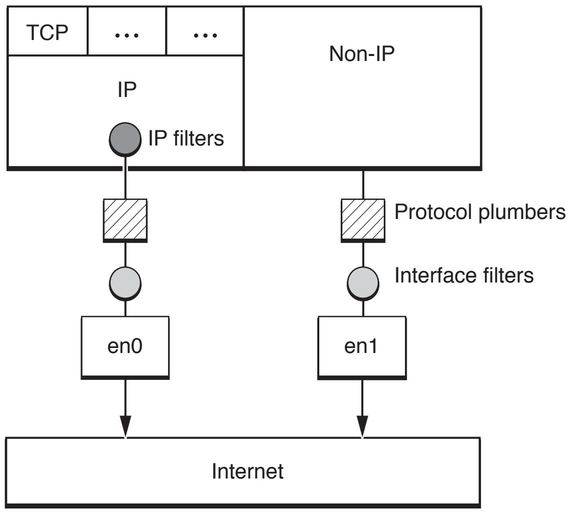
It is possible to create filter loops in which one filter changes a value and reinjects the packet, which causes a second filter to change the value back and reinject it in an endless loop.

To reduce the likelihood of such a loop, when reinjecting packets, your filter should always specify itself as the `filter_ref` parameter.

Interface Filters

This chapter describes the programming interface for creating interface filters, which are associated with a particular network interface, as shown in Figure 6-1 (page 37).

Figure 6-1 Data Link Interface Layer



Interface Filter Functions and Callbacks

An interface filter defines the following callbacks:

- `iff_input_func`, which is used to process packets after they are demuxed.
- `iff_output_func`, which is used to process packets before they are sent out from an interface to a network.
- `iff_event_func`, which is used to filter events specific to an interface.
- `iff_ioctl_func`, which is used to filter ioctls sent to an interface. All undefined ioctls are reserved for future Apple use. You should use `kern_control` if you need to add additional control mechanisms.
- `iff_detached_func`, which is called when your filter is detached from an interface. When this callback is called, you should perform any cleanup related to the interface. If this is called as a result of the interface itself being detached, it will occur after you receive the interface detach notification.

To attach and detach an interface filter, the following functions are defined by the interface filter KPI:

- `iflt_attach`, which inserts an interface filter between the protocol plumbers and an attached interface.
- `iflt_detach`, which removes a previously inserted interface filter.

Common Caveats

There are a number of surprises that you may run into when writing an interface filter. Several of these follow:

Packet injection

When your filter injects packets, it should use the `ifnet_input` and `ifnet_output_raw` functions. If you do this, your filter should be prepared to ignore the packet it just injected, as your filter's `iff_input_func` or `iff_output_func` callback will see this packet again immediately. You should use the `mbuf_tag` APIs (`mbuf_tag_allocate`, for example) to track these packets. If multiple filters are swallowing and reinjecting packets, you may see a given packet multiple times.

Note: When reinjecting packets, the filter must ensure that the packet header field is set in the first mbuf structure. Otherwise, the call to `ifnet_input` will result in a kernel panic (`NULL` pointer dereference).

When your `iff_input_func` callback is called, you may find that the `packet_header` field has been set to `NULL`. The `frame_ptr` parameter to `iff_input_func` can be used to set the `packet_header` field if the packet must be reinjected. To do this, use the `mbuf_pkthdr_setheader` function to set the `packet_header` field in the mbuf.

If your `iff_input_func` callback does not swallow a packet, it is not necessary to set the `packet_header` field.

Input callbacks: Header pointers and mbufs

Your filter's input callback receives an mbuf pointer to the packet contents and a separate header pointer. The header pointer references the link-layer header, as defined by the relevant interface.

For most interfaces, the length of this header can be determined by inspecting the header length (`ifnet_hdrlen`) defined by the interface. For some interfaces, however, such as PPP, the header length is variable.

Output callbacks: Header pointers and mbufs

Your filter's output callback receives the entire packet in the mbuf chain. To get the protocol layer information, your filter must know how to parse the link-layer header. For this reason, if you are writing a filter that needs to work with IP packets, you should consider writing an IP filter unless it is absolutely necessary to access link-layer information.

Network Interfaces and Protocol Plumbers

This chapter describes the network interface KPI. This programming interface allows a KEXT to attach new network interfaces, communicate with and manipulate network interfaces, and create new virtual interfaces.

The mechanism recommended for supporting new interfaces depends on the nature of the interface. The recommended mechanisms are:

- Ethernet drivers—subclass the `IOEthernetController` and `IOEthernetInterface` classes from the I/O Kit's I/O Networking Family.
- Other hardware network controllers—subclass the `IONetworkController` and `IONetworkInterface` classes.
- Virtual interfaces—the interface KPI described in this section is recommended.

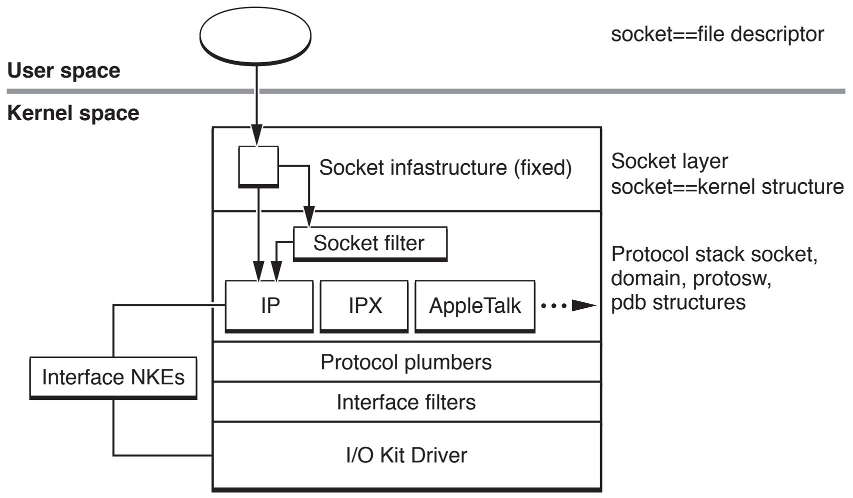
This chapter also describes protocol plumbers. Protocol plumbers are used to attach network protocols to interfaces.

If you are creating support for an interface type that the stack does not already support (such as ATM), regardless of whether your KEXT uses the I/O Kit, you must register protocol plumbers for attaching existing protocols to the new interface type.

Note: Since more than one developer might attempt to register plumbers for a given interface type, your KEXT should be prepared to handle such a situation. If the demux descriptors are standard, it should be possible for your KEXT to work with a third-party plumber.

The functionality in a network interface is utilized by both the interface driver and the protocol stacks, as shown in Figure 7-1 (page 39).

Figure 7-1 Network Interfaces in the Networking Stack



Network Interface Callbacks

Your network interface should define the following callbacks, which are called by protocols and drivers:

- `ifnet_add_proto_func`, which is called whenever a protocol is attached to the interface.
- `ifnet_check_multi`, which is called when a multicast address is added to an interface. This allows the interface to reject invalid multicast addresses before they are added to the interface.
- `ifnet_del_proto_func`, which is called when a protocol is detached from the interface.
- `ifnet_demux_func`, which is called with a raw packet from the interface, and returns the protocol family value of the protocol that should process the packet. It can do this using either the demux descriptors registered with `ifnet_add_proto_func` or hard-coded logic. If the packet does not match any protocol, the function should return `ENOENT`.
- `ifnet_detached_func`, which is called when your interface is detached.
- `ifnet_event_func`, which is called when an event occurs on a particular interface.
- `ifnet_framer_func`, which is called with outgoing packets before sending them to outgoing interface filters, and is expected to wrap the packet with a stack frame appropriate to the interface type.
- `ifnet_ioctl_func`, which is called whenever an ioctl is received for the interface. The network interface is expected to pass these on to the I/O Kit driver if it is necessary and appropriate to do so. All undefined ioctls are reserved for future Apple use. You should use `kern_control` if you need to add additional control mechanisms.
- `ifnet_output_func`, which is called with a packet that is ready to be sent out the wire. The network interface is expected to transmit the packet and free the mbuf associated with it. For network interfaces backed by I/O Kit drivers, this callback generally calls a function in the I/O Kit driver that handles both of these tasks.
- `ifnet_set_bpf_tap`, which is called by the stack to set the BPF tap function that is installed on the interface. This callback is optional, but recommended; if you do not add this function, BPF cannot be used with your interface.

Note: To support BPF, in addition to defining an `ifnet_set_bpf_tap` callback, your KEXT must also call `bpfattach` after attaching the interface to the stack.

Good examples of many of these functions can be found in `bsd/net/ether_if_module.c` in the `xnu` (kernel) source tree.

Installing and Removing Network Interfaces

The following functions are typically called (in the following order) to support the dynamic insertion and removal of network interfaces:

1. `ifnet_allocate`, which allocates an interface structure.
2. `ifnet_attach`, which attaches an interface to the global interface list.

3. `bpfattach`, which enables BPF support on an interface (optional).
4. `ifnet_detach`, which removes an interface from the global interface list.
5. `ifnet_release`, which releases a reference to an interface structure. If the reference count reaches zero (0), the structure will be freed. This release matches the `ifnet_allocate` call earlier, and should be called *after* calling `ifnet_detach`.

The related function `ifnet_reference` can be used (generally by other KEXTs) to increase the reference count of an interface structure. These calls must be balanced by an equal number of calls to `ifnet_release`.

Protocol Plumbers

Protocol plumbers, as mentioned previously, are responsible for attaching a protocol to a network interface. When a protocol needs to attach to an interface, it calls a function that looks up the plumber designated for that protocol and interface type, then calls that plumber's plumb handler.

Protocol plumbers define the following callbacks, which are called by protocols:

- `proto_plumb_handler`, which is called to attach a protocol to an interface. This typically consists of a call to the interface's `ifnet_attach_protocol` callback.
- `proto_unplumb_handler`, which is called to detach a protocol from an interface. The unplumb handler should call the `ifnet_detach_protocol` function, but may do other cleanup such as freeing any storage allocated in the `proto_plumb_handler` callback.

Note: The protocol unplumb handler is optional. If it is `NULL`, the stack will directly call `ifnet_detach_protocol` to unplumb the protocol. The plumber only needs to specify an unplumb handler if it needs to do additional cleanup.

When attaching a protocol to an interface, the protocol plumber typically fills in the following fields in the `ifnet_attach_proto_param` structure, many of which may be defined as part of the protocol itself. This mechanism provides the opportunity for the protocol plumber to intercept these calls and take interface-specific actions where needed.

- `proto_media_detached` (optional), which is used to notify the protocol that it is being detached.
- `proto_media_event` (optional), which is called to notify a protocol about interface-specific events.
- `proto_media_input` (required), which is used to deliver an inbound packet to the protocol for processing.
- `proto_media_ioctl` (optional), which is typically the protocol's ioctl handling function.
- `proto_media_preout` (required), which is called just before a packet is transmitted. This allows the protocol to specify a media-specific frame type and destination.
- `proto_media_resolve_multi` (optional), which is used to obtain a link layer address for a given protocol layer multicast address. This is only necessary if your interface supports multicast.
- `proto_media_send_arp` (optional), which is used to obtain the link layer address corresponding to a given protocol layer unicast address. This is necessary for all non-point-to-point interfaces.

Examples of these functions can be found in `bsd/net/ether_inet_pr_module.c` in the `xnu` (kernel) source tree.

Sending Data

The following steps describe the process of sending a packet, using Ethernet as the example medium:

1. The `ip_output` routine in the IP protocol stack calls `ifnet_output`.
2. The `ifnet_output` function calls the protocol plumber's `proto_media_preout` function. In the case of IP, this function calls `inet_arp_lookup`.
3. If the ARP cache does not contain an entry for the IP address, `inet_arp_lookup` then calls the protocol plumber's `proto_media_send_arp` callback to resolve the target IP address into a media access control (MAC) address.
4. When the `proto_media_preout` callback returns, the `ifnet_output` function calls the network interface's `ifnet_framer_func` function. This framing function prepends interface-specific frame data to the packet.
5. If any interface filters are present, their `iff_output_func` callbacks are called consecutively.
6. The `ifnet_output` function calls the network interface's `ifnet_output_func` callback, which transmits the packet and frees the mbuf.

Receiving Data

The following steps describe the process of receiving a packet:

1. The hardware driver or its support code calls `ifnet_input` with pointers to its `ifnet` structure (`ifnet_t`) and `mbuf` chain (`mbuf_t`).
2. The packet is queued. Processing resumes on a different thread.
3. The `ifnet_input` function calls the network interface's `ifnet_demux_func` function for the interface.
4. The demultiplexing function identifies the frame and returns a `protocol_family_t` value to indicate which protocol should handle the packet.
5. The `ifnet_input` function calls the attached interface filters (if any) sequentially.
6. Any packets not matching an attached protocol are dropped, as are any promiscuous packets.
7. The `ifnet_input` function calls the protocol plumber's `proto_media_input` function. The plumber is specific to a given protocol/interface combination.

Note: The Ethernet-specific module for IP receives the frame and delivers the packet to the protocol's `proto_input` routine.

Network Kernel Extensions Reference

The document *KPI Reference* describes the functions that NKEs can call and describes a few NKE-specific data types. They are organized by header file. This chapter includes a few additional APIs that may be useful when writing an NKE.

- "Kernel Utilities" (page 43) lists some kernel utilities that NKEs can call.
- `kpi_interface.h` includes functions for manipulating network interfaces, including packet injection, attaching/detaching protocols, attaching/detaching interfaces, and so on.
- `kpi_interfacefilter.h` includes functions for filtering at the raw packet level, just above the network interface layer. These functions are appropriate for an interface filter.
- `kpi_ipfilter.h` includes functions for attaching a packet filter for IPv4 or IPv6 packets. These functions are appropriate for a KEXT that filters IP traffic.
- `kpi_mbuf.h` includes functions for manipulating mbuf data structures. These are used heavily for passing packets and packet fragments around throughout the protocol stack.
- `kpi_protocol.h` includes functions for packet injection. It also includes functions to register "plumbers"—handlers that deal with requests for attaching a protocol to an interface (and detaching, and so on).
- `kpi_socket.h` includes functions for manipulating a socket, including packet send/receive and flag manipulation.
- `kpi_socketfilter.h` includes functions and data type definitions for creating a socket filter.

Kernel Utilities

NKEs can call a number of kernel utility functions including the following:

- "`_MALLOC`" (page 44)
- "`_FREE`" (page 44)
- "`printf`" (page 44)
- "`proc_exiting`" (page 45)
- "`proc_is64bit`" (page 45)
- "`proc_rele`" (page 45)
- "`proc_self`" (page 45)
- "`proc_suser`" (page 45)
- "`msleep`" (page 45)
- "`wakeup`" (page 46)

- "wakeup_one" (page 46)

This list does not attempt to be exhaustive, but highlights many of the more frequently-used utility functions.

_MALLOC

Defined in: `<sys/malloc.h>`

Allocates kernel memory.

```
void *_MALLOC(size_t size, int type, int flags);
```

`_MALLOC` is much like the user-space `malloc` function, but it has additional parameters that require some explanation.

The `types` argument is a number representing the type of data that will be stored in the argument. This is used primarily for accounting purposes. The known types are described in `<sys/malloc.h>`.

The flags argument consists of some combination of `M_WAITOK`, `M_NOWAIT`, and `M_ZERO`.

The flag `M_NOWAIT` causes `_MALLOC` to immediately return a null pointer if no space is available rather than waiting for space to become available. While this is appropriate for time-sensitive tasks that can be retried, it is not always what you want.

The more traditional (and default) behavior is `M_WAITOK`, which indicates that it is safe to wait for space to become available. If your code is in a critical path for performance, you should probably use `M_NOWAIT` if possible, and depend on the networking stack to retry after resources become available.

Finally, the flag `M_ZERO` requests that the allocator should zero the resulting allocation before returning it.

_FREE

Defined in: `<sys/malloc.h>`

Frees memory allocated with `_MALLOC`

```
void _FREE(void *addr, int type);
```

The `type` flag passed to `_FREE` must be the same as the flag passed to the corresponding call to `_MALLOC`.

printf

Defined in: `<libkern/libkern.h>`

Print text to the console.

```
void printf(const char *format, ... );
```

Identical to `printf` in user space. It is not safe to call `printf` from within an interrupt context. This should generally not be an issue, as you should avoid calling NKE functions from within an I/O Kit driver's filter routine as a matter of course, but it is worth noting.

proc_exiting

Defined in: `<sys/proc.h>`

Returns 1 if a process is exiting, else 0.

```
int proc_exiting(proc_t);
```

proc_is64bit

Defined in: `<sys/proc.h>`

Returns 1 if a process is a 64-bit executable, else 0.

```
int proc_is64bit(proc_t);
```

proc_rele

Defined in: `<sys/proc.h>`

Releases a reference to a process entry.

```
int proc_rele(proc_t);
```

proc_self

Defined in: `<sys/proc.h>`

Returns a reference to the running process. This must be released with "`proc_rele`" (page 45).

```
proc_t proc_self(void);
```

proc_suser

Defined in: `<sys/proc.h>`

Checks to see if a process is running as the super-user (root).

```
int proc_suser(struct proc *p);
```

Note: There are many other `proc_*` functions available. These are described in `<sys/proc.h>`. The ones here are just a few of the more commonly used functions in network-related KEXTs.

msleep

Defined in: `<sys/proc.h>`

Sleep until an event is posted with "wakeup" (page 46) or until a timeout occurs. This is commonly combined with a timeout value to bound the wait.

```
    int msleep(void *chan, lck_mtx_t *mtx, int pri, const char *wmesg, struct
timespec *ts);
```

The timeout value is a standard timespec (defined in `<sys/time.h>`, and is measured in seconds and nanoseconds. To sleep until woken, you should pass a `NULL` value for `ts`.

Note: For compatibility reasons, an alternate form, `msleep1`, is provided, in which the last argument is a 64-bit deadline in Mach abstime format. This form can be substituted in place of the `tsleep` function used in pre-KPI network kernel extensions by using `clock_interval_to_deadline` to obtain a mach abstime deadline from the time interval.

The parameter `mtx` is a mutex (defined in a processor-specific include, but included with `<sys/lock.h>`) that will be released prior to sleeping. (The mutex *must* be locked prior to calling msleep.) The mutex will also be reacquired upon wake unless the `PDROP` flag is set in the priority value.

Note: If the `PDROP` flag is specified, `msleep` returns with the mutex unlocked regardless of whether it actually blocks or not.

The parameter `chan` should be a unique identifier specific to a given wait event. Usually such an event is associated with the change in a variable, in which case the address of that variable makes a good value for `chan`.

The parameter `pri` is the desired priority on wake (defined in `<sys/param.h>`). After another thread has called "wakeup" (page 46) on the desired event (specified by the value of `chan`), your code will begin executing at the specified priority. If the `PCATCH` flag is set on `pri`, signal handlers will be tried before and after the sleep.

Returns 0 if awakened with wakeup, `EWOULDBLOCK` on timeout expiry, and `ERESTART` or `EINTR` if `PCATCH` is set and a signal occurred, depending on whether the `SA_RESTART` flag is set on the signal.

wakeup

Defined in: `<sys/proc.h>`

Wakes all threads sleeping on a given channel through a call to "msleep" (page 45).

```
    void wakeup(void *chan);
```

wakeup_one

Defined in: `<sys/proc.h>`

Wakes the first thread sleeping on a given channel through a call to "msleep" (page 45).

```
    void wakeup_one(void *chan);
```

Glossary

domain A complete protocol family.

driver layer I/O Kit Drivers for various networking types.

extension A general term for an object module that can be dynamically added to a running system; often used as a synonym for kernel extension.

global socket filter A socket filter that is automatically enabled for sockets of the type specified.

ifnet structure A data structure containing function pointers and data related to a particular network interface.

in-band Communication on a socket or interface that contains actual data destined for the endpoint (for example, `send` and `recv` calls). See also: out-of-band.

interface filter A filter that attached to a particular interface. An interface filter alters in-band and out-of-band communication specific to a given interface.

interface layer A layer above the driver layer containing interface KEXTs, interface filters, and protocol plumbers.

interface KEXT A network kernel extension that provides routines specific to a particular family of interfaces, such as ARP equivalence routines.

IP filter A filter that alters IP traffic each time it enters the protocol stack. By its very nature, an IP filter can only filter in-band communication.

KEXT Short for kernel extension; a plug-in for the Mac OS X kernel (xnu).

KPI Short for kernel programming interface; a group of opaque data types and accessor functions designed to maintain binary compatibility across OS releases.

mbuf A data structure containing data about a network packet.

network kernel extension (NKE) 1) The architecture that allows modules to be added to the Mac OS X networking subsystem while the system is running. 2) A module that can be added to a running system.

out-of-band Communication on a socket or interface that relates to the operation of the socket or interface rather than data destined for the endpoint (for example, `ioctl` and `getsockopt` calls). See also: in-band.

plug-in A general term for an object module that can be dynamically added to a running system.

programmatic socket filter A socket filter that is enabled only under program control by calling `setsockopt` on a specific socket.

protocol plumber A network kernel extension that routes data between an interface and a network protocol stack.

protocol stack A layer of the kernel network architecture containing the core functionality for a protocol family such as TCP/IP.

protosw structure A data structure containing function pointers and data associated with a protocol family.

socket structure A data structure containing data associated with a network socket.

socket filter A filter that is associated with a particular socket or class of sockets, filtering in-band and out-of-band operations on the socket. A socket filter resides between a socket and the protocol layer.

Document Revision History

This table describes the changes to *Network Kernel Extensions Programming Guide*.

Date	Notes
2009-08-14	Changed links from KPI Reference to Kernel Framework Reference.
2009-03-02	Fixed an error in a code listing related to the return value of the socket function.
2008-04-08	Added additional details about msleep function.
2007-05-03	Added art to mbuf chapter.
2007-04-03	Added socket KPI conceptual material.
2006-10-03	Fixed a minor error in a code example.
2005-08-11	Added additional note clarifying the relationship between CTL_FLAG_REG_SOCK_STREAM and SOCK_STREAM.
2005-06-04	Fixed minor typographical errors.
2005-04-29	Updated for Mac OS X 10.4 and KPI interfaces.
2004-04-22	Initial republication (for Mac OS X 10.3 and prior)