Network Kernel Extensions (legacy)

Darwin > Kernel

2006-10-03

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Figures and Listings

Chapter 1 About Network Kernel Extensions 9

About Network Kernel Extensions

Important: The information provided in this document is relevant for Mac OS 10.1 through 10.3. Specific mention is made for items which apply to newer releases of the OS.

An important change has long been noted in the <sys/mbuf.h> header file since the release of Mac OS X 10.2. Note that the header file is bracketed by the __APPLE_API_UNSTABLE define. The mbuf structure is a key to the processing of packets in an NKE. As part of the formalizing the NKE APIs, it is expected that the mbuf structure will be changed. Details will be provided in the future. Changes to the existing NKE API are not expected be applied to System Updates to Mac OS X 10.3.x, however, bug fixes or features for future systems may require some interim changes.

For all shipping releases of Mac OS X prior to 10.4, the Network Kernel Extensions (NKE) APIs have not been officially supported. The legacy NKE architecture was implemented as an interim solution. The legacy API was never designed to be officially supported. Other aspects of the OS X networking implementation have received a higher priority, and so the interim solution has remained in effect to OS X 10.3.x.

The NKE mechanism for Mac OS X version 10.4 and later is described in the document *Network Kernel Extensions Programming Guide*.

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

NKEs allow you to

- create protocol stacks that can be loaded and unloaded dynamically and configured automatically.
- create modules that can be loaded and unloaded dynamically at specific positions in the network hierarchy. These modules can monitor network traffic, modify network traffic, and receive notification of asynchronous events at the data link and network layers from the driver layer, such as power management events and interface status changes.

An NKE is a specific case of a Mac OS X kernel extension. It is a separately compiled module (produced, for example, by XCode using the Kernel Extension project type).

An installed and enabled NKE is invoked automatically, depending on its position in the sequence of protocol components, to process an incoming or an outgoing packet. Loading (installing) 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 only possible for IOKit extensions and other extensions that IOKit extensions depend on.

As a kernel extension, an NKE provides initialization and termination routines that the Kernel Extension Manager invokes when it loads or unloads an NKE. The initialization routine handles any operations needed to complete the incorporation of the NKE into the kernel, such as updating `protosw` and `domain` structures. Similarly, the termination routine must remove references to the NKE from these structures in order to unload itself successfully. NKEs must provide a mechanism, such as a reference count, to ensure that the NKE can terminate without leaving dangling pointers.

NKE Implementation

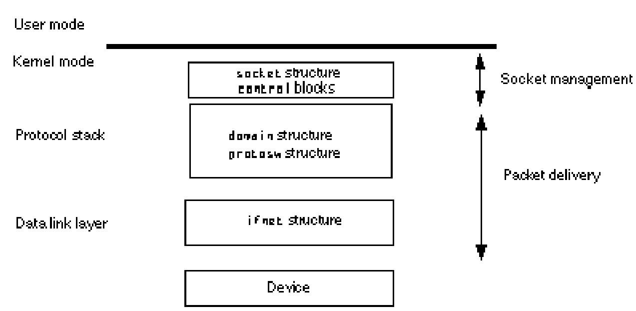
Review of 4.4 BSD Network Architecture

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 4.4BSD network architecture



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

NKE Types

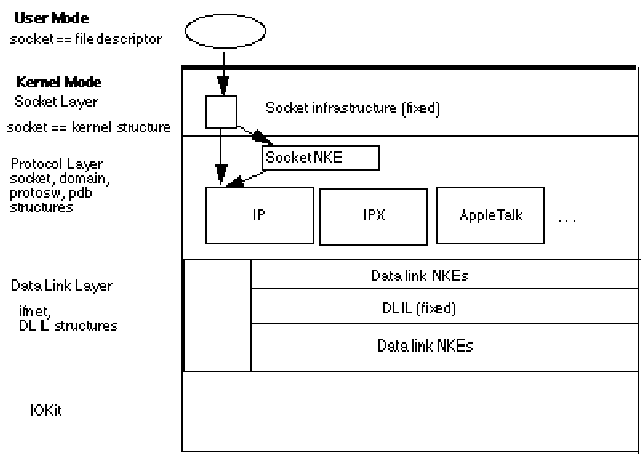
Making the 4.4BSD network architecture dynamically extensible requires several NKE types that are used at specific locations within the kernel.

- socket NKEs, which reside between the network layer and protocol handlers and are invoked through a `protosw` structure. Socket NKEs use a new set of override dispatch vectors that intercept specific socket and socket buffer utility functions.
- protocol family NKEs, which are collections of protocols that share a common addressing structure. Internally, a `domain` structure and a chain of `protosw` structures describe each protocol.

- protocol handler NKEs, which process packets for a particular protocol within the context of a protocol family. A `protosw` structure describes a protocol handler and provides the mechanism by which the handler is invoked to process incoming and outgoing packets and for invoking various control functions.
- data link NKEs, which are inserted below the protocol layer and above the network interface layer. This type of NKE can passively observe traffic as it flows in and out of the system (for example, a sniffer) or can modify the traffic (for example, encrypting or performing address translation). Data link NKEs can provide media support functions (performing demultiplexing, framing, and pre-output functions, such as ARP) and can act as "filters" that are inserted between a protocol stack and a device or above a device.)

Figure 1-2 (page 11) summarizes the NKE architecture.

Figure 1-2 NKE architecture



Global and Programmatic NKEs

Socket NKEs can operate in one of two modes: programmatic or global.

A global NKE is an NKE that is automatically enabled for sockets of the type specified for the NKE.

A programmatic NKE is a socket NKE that is enabled only under program control, using socket options, for a specfic socket.

Data link `filters' are essentially global in that they can't be accessed by specific sockets.

Tracking NKE Usage

To support the dynamic addition and removal of NKEs in Mac OS X, the kernel keeps track of the use of NKEs by other parts of the system.

Use of protocol family NKEs is tracked by the `dom_refs` member of the `domain` structure, which has been added to support NKEs in Mac OS X. The kernel's `socreate` function increments `dom_refs` each time `socreate` is called to create a socket in an NKE domain. The `socreate` function is called when user-mode applications call `socket` or when `sonewconn` successfully connects to a local listening socket. The `dom_refs` member is decremented each time `soclose` is called to close a socket connection.

Use of protocol handler NKEs is tracked by the pr_refs member of the protosw structure, which has been added to support NKEs in Mac OS X. Like the dom_refs member of the domain structure, the pr_refs member of the protosw structure tracks the use of the protocol between calls to socreate and sonewconn to create a socket and soclose to close a socket.

The most important aspect of removing an NKE is ensuring that all references to NKE resources are eliminated and that all system resources allocated by the NKE are returned to the system. The NKE must track its use of resources, such as socket structures and protocol control blocks, so that the NKE's termination routine can eliminate references and return system resources.

Modifications to 4.4BSD Networking Architecture

To support NKEs in Mac OS X, the 4.4BSD domain and protosw structures were modified as follows:

- The protosw array referenced by the domain structure is now a linked list, thereby removing the array's upper bound. The new dom_maxprotohdr member defines the maximum protocol header size for the domain. The new dom_refs member is a reference count that is incremented when a new socket for this address family is created and is decremented when a socket for this address family is closed.
- The protosw structure is no longer an array. The pr_next member has been added to link the structures together. This change has implications for protox usage for AF_INET and AF_ISO input packet processing. The pr_flags member is an unsigned integer instead of a short. NKE hooks have been added to link NKE descriptors together (pr_sfilter).

PF_NKE Domain

Mac OS X defines a new domain -- the PF_SYSTEM domain-- whose purpose is to provide a way for applications to configure and control NKEs. The PF_SYSTEM domain has two protocols, of which only one is of interest for communications with the NKE:

- The SYSPROTO_CONTROL protocol is used for configuring and controlling all NKEs.

Internally, the PF_SYSTEM domain's initialization function is called when the PF_SYSTEM domain is initially added to the system. The initialization function adds the SYSPROTO_CONTROL protocol to the domain's protosw list and performs other initialization tasks.

In the NKE's start method, register a Kernel Controller structure using the ctl_register function. The ctl_register function is defined in <sys/kern_control.h>. The ctl_register call is prototyped as follows.

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

The fields of the kern_ctl_reg structure are defined as follows.

ctl_id - unique 4 byte id for the controller. Enter a registered Creator ID. Go to the Apple Developer Creator ID web page to register a unique ID. See http://developer.apple.com/dev/cftype/ for more information.

ctl_unit - the unit number for the controlller. A controller can be registered multiple times with the same ctl_id, but for each instance and different unit number must be used.

`ctl_flags` - set to `CTL_FLAG_PRIVILEGED` which requires that the user must have admin privileges to contact the controller.

`ctl_sendsize` - size of buffer reserved for sending messages. 0 = default value.

`ctl_recvsize` - size of buffer reserved for receiving messages. 0 = default value.

Dispatch Functions

ctl_connect - called when the client process calls connect on the socket with the id/unit number of the registered controller.

clt_disconnect - called when the user client process closes the control socket.

ctl_write - called when the user client process writes data to the socket.

ctl_set - called when the user client process setsockopt to set the controller configuration.

ctl_get - called when the user client process calls getsockopt on the socket.

The following is a code example of this process.

Listing 1-1 Dispatch example

```
struct kern_ctl_reg     ep_ctl;
// Initialize controller
bzero(&ep_ctl, sizeof(ep_ctl));  // sets ctl_unit to 0
ep_ctl.ctl_id = kEPCommID; // should be unique -
                                   // use a registered Creator ID here
ep_ctl.ctl_flags = CTL_FLAG_PRIVILEGED;
ep_ctl.ctl_write = EPHandleWrite;
ep_ctl.ctl_get = EPHandleGet;
ep_ctl.ctl_set = EPHandleSet;
ep_ctl.ctl_connect = EPHandleConnect;
ep_ctl.ctl_disconnect = EPHandleDisconnect;
error = ctl_register(&ep_ctl, &gEPState, &gEPState.ctlHandle);


int EPHandleSet( kern_ctl_ref ctlref, 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;
}

int EPHandleGet( kern_ctl_ref ctlref, 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;
}

int
EPHandleConnect(kern_ctl_ref ctlref, void *userdata)
{
#if DO_LOG
    log(LOG_ERR, "EPHandleConnect called\n");
#endif
    return (0);
}

void
EPHandleDisconnect(kern_ctl_ref ctlref, void *userdata)
{
#if DO_LOG
    log(LOG_ERR, "EPHandleDisconnect called\n");
#endif
    return;
}

int EPHandleWrite(kern_ctl_ref ctlref, void *userdata, struct mbuf *m)
{
#if DO_LOG
    log(LOG_ERR, "EPHandleWrite called\n");
#endif
    return (0);
}
```

Connection from the Client Process

After the NKE registers a Kernel Controller structure the application level process opens a PF_SYSTEM socket. The application level process sets up the sockaddr_ctl structure with the required parametrs to communicate with the NKE's Kernel Controller.

To communicate with the NKE, the client process opens a `PF_SYSTEM` socket using the socket call.

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

The client process uses the connect call with the file descriptor returned from the socket call to establish a connection with the NKE. In making the connect call, fill in the sockaddr_ctl structure as follows.

```
sc_len = sizeof(struct sockaddr_ctl);
sc_family = AF_SYSTEM;
ss_sysaddr = AF_SYS_CONTROL;
```

```
sc_id = set to value of ctl_id registered by the NKE in the ctl_reguster call
described above.
sc_unit = set to the unit number registered by the NKE in the ctl_register call
 described above.
```

The client process uses the setsockopt call to send commands to the NKE. Note that the option names are user defined. The NKE defines what option names it will respond to, and the client process must pass only supported option names to the NKE in the setsockopt call.

The client process uses the getsockopt call to get status information from the NKE. Note that the option names are user defined. The NKE defines what option names it will respond to, and the client process must pass only supported option names to the NKE in the setsockopt call.

The following is a code example for opening a PF_SYSTEM socket to communicate with an NKE

Listing 1-2 Opening a `PF_SYSTEM` socket

```
      struct sockaddr_ctl       addr;
      int                       ret = 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;
      addr.sc_id = kEPCommID;  // should be unique - use a registered Creator ID here

      fd = socket(PF_SYSTEM, SOCK_DGRAM, SYSPROTO_CONTROL);
      if (fd)
      {
        result = connect(fd, (struct sockaddr *)&addr, sizeof(addr));
        if (result)
           fprintf(stderr, "connect failed %d\n", result);
      }
      else
        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);
       etc.
```

Implementing a Preference File for NKE

The question arises as to how an NKE can open a "preference file" in the start method. Under the existing architecture, the NKE cannot reliably access a Preference File. When the system starts the NKE, there are no APIs, which the NKE can use to open a file and read preference information. While the NKE could access its info.plist, there is the assumption that the info.plist will not be changed across startups as this information is cached by the system in order to expedite startups.

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 communication method described above, and passes in configuration information that the NKE may require.

About Protocol Family NKEs

Adding and removing protocol family NKEs is accomplished by calling `net_add_domain` and `net_del_domain`, respectively. These calls are described in "Protocol Family NKE Functions" (page 40). For detailed information about implementing protocol families, see `The Design and Implementation of the 4.4 BSD Operating System` by M. K. McKusick. et al. and `TCP/IP Illustrated` by Richard W. Stevens.

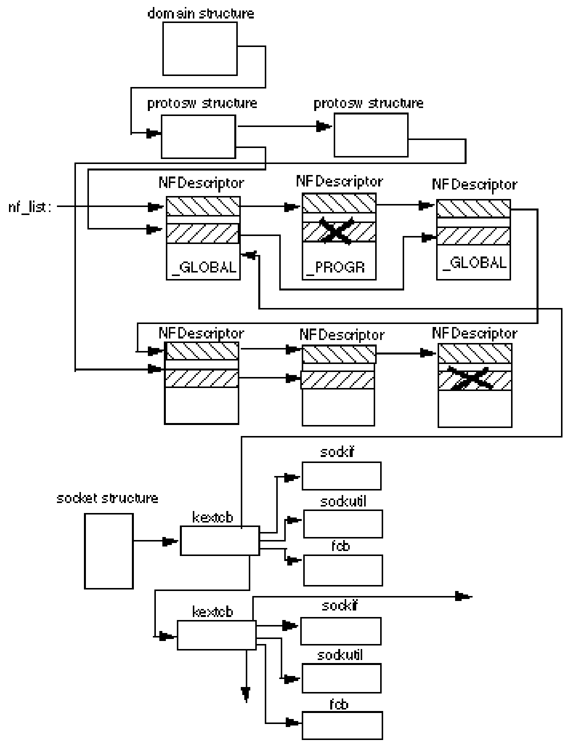
About Protocol Handler NKEs

Adding and removing protocol handler NKEs is accomplished by calling `net_add_proto` and `net_del_proto`, respectively. These calls are described in "Protocol Handler NKE Functions" (page 41). For detailed information about implementing protocol families, see `The Design and Implementation of the 4.4 BSD Operating System` by M. K. McKusick. et al. and `TCP/IP Illustrated` by Richard W. Stevens.

About Socket NKEs

Socket NKEs are installed in the kernel by calling `register_sockfilter` typically from the NKE's initialization routine. Each socket NKE provides a descriptor structure that is linked into a global list (`nf_list`). A second chain runs through the filter descriptor to link it to a `protosw` for global NKEs. Figure 1-3 (page 17) shows the interconnections for these data structures.

Figure 1-3 Domain structure and protosw interconnections



When you call `socreate` to create a socket, any global NKEs associated with the corresponding `protosw` structure are attached to the socket structure using the `so_ext` field to link together `ketcb` structures that are allocated when the socket is created. (See Figure 1-3 (page 17).) These `ketcb` structures are initialized to point to the extension descriptor and two dispatch vectors of intercept functions (one for socket operations and one for socket buffer utilities).

The filter descriptor for a programmatic NKE is linked into the `nf_list` in the same way as are global NKEs but the file descriptor does not appear in the list associated with a `protosw`. A program can call `setsocketopt` using socket option `SO_NKE`) to insert a programmatic NKE into its NKE chain in the same way that it would call `setsocketopt` to insert a global NKE.

Each socket NKE has two dispatch vectors, a `sockif` structure and a `sockutil` structure, that contain pointers to the NKE's implementation of these functions. The functions are called when the corresponding `socket` and `sockbuf` functions are are called. The dispatch vectors permit the NKE to selectively intercept socket and socket buffer utilities. Here is an example:

```
int (*sf_sobind)(struct socket *, struct mbuf *, st kextcb);
```

The kernel's `sobind` function calls the NKE's `bind` entry point with the arguments passed to `sobind` and the `kextcb` pointer for the NKE. The `sockaddr` structure contains the name of the local endpoint being bound.

Each of the intercept functions can return an integer value. A return value of zero is interpreted to mean that processing at the call site can continue. A non-zero return value is interpreted as an error (as defined in `<sys/errno.h>`) that causes the processing of the packet or opertation to halt. If the return value is

EJUSTRETURN, the calling function (for example, sobind) returns at that point with a value of zero. Otherwise, the function returns the non-zero error code. In this way, an NKE can "swallow" a packet or an operation. An NKE may reinject the packet at a later time. (Note that the injection mechanism is not yet defined.)

A program can insert a socket NKE on an open socket by calling setsockopt 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 nke_handle specifies the NKE to be linked to the socket (with the so_ext link). It is the programmer's task to locate the appropriate NKE, assure that it is loaded, and retain the returned handle for use in the setsockopt call.

The nke_where value specifies an NKE assumed to be in this linked list. If nke_where is NULL, the NKE represented by nke_handle is linked at the beginning or end of the list, depending on the value of nke_flags.

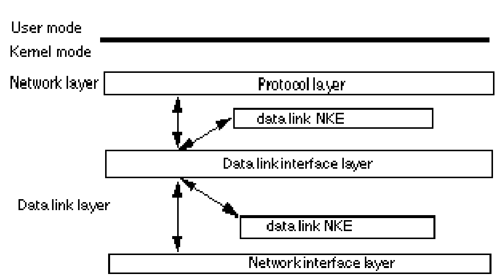
The nke_flags value specifies where, relative to nke_where, the NKE represented by nke_handle will be placed. Possible values are NFF_BEFORE and NFF_AFTER defined in <net/kext_net.h>.

The nke_handle and nke_where 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.

About Data Link NKEs

This section describes the programming interface for creating data link NKEs, which are inserted below the protocol layer and above the network interface layer. Data link NKEs depend on the Data link interface layer (DLIL), shown in Figure 1-4 (page 18), which provides a fixed point for the insertion of data link NKEs.

Figure 1-4 Data Link Interface Layer



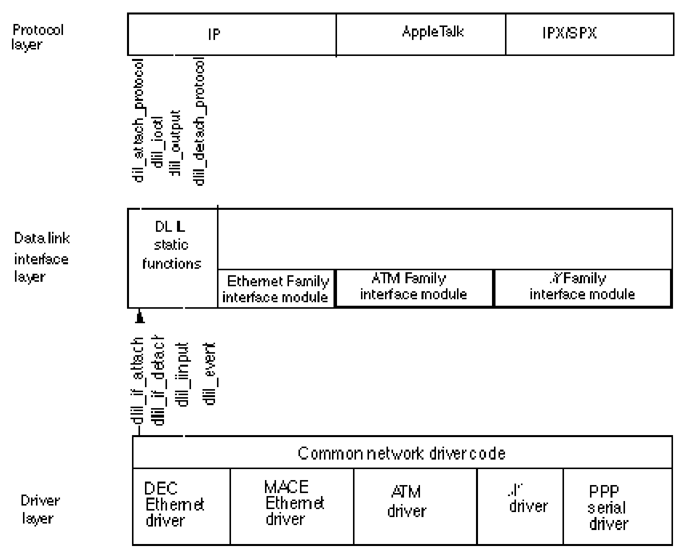
DLIL Static Functions

The DLIL defines the following static functions, which are called by protocols and drivers:

- `dlil_attach_protocol`, which attaches network protocol stacks to specific interfaces
- `dlil_detach_protocol`, which detaches network protocol stacks from the interfaces to which they were previously attached
- `dlil_if_attach`, which registers network interfaces with the DLIL
- `dlil_if_detach`, which deregisters network interfaces that have been registered with the DLIL
- `dlil_ioctl`, which sends ioctl commands to a network driver
- `dlil_input`, which sends data to the DLIL from a network driver
- `dlil_output`, which sends data to a network driver
- `dlil_event`, which processes events from other parts of the network and from IOKit components. (Note that the event mechanisms are still under development.)

In Figure 1-5 (page 19), the DLIL static functions are shown in relation to the DLIL, the protocol layer, and the network driver layer.

Figure 1-5 DLIL static functions



Changes to the ifnet and if_proto Structures

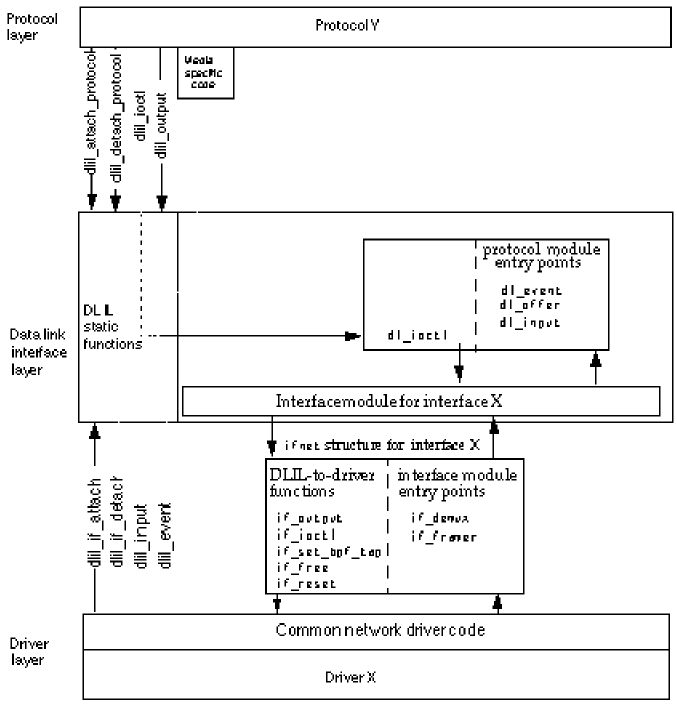
To support data link NKEs, the traditional `ifnet` structure as been extended in Mac OS X: the driver or software that supports the driver must allocate a separate `ifnet` structure for each logical interface. When an interface is attached (by calling `dlil_if_attach`)to the DLIL, the DLIL receives a pointer to that interface's `ifnet` structure.

Each interface can transmit and receive packets for multiple network protocol families, so for each attached protocol family the DLIL creates an `if_proto` structure chained off the `ifnet` structure for that interface.

The `if_proto` structure contains function pointers that the DLIL uses to pass incoming packets and event information to the protocol stack, as well as a pointer to the protocol dependent "pre-output" function that performs protocol-family specific operations such as network address translation on outbound packets.

Figure 1-6 (page 20) shows the `ifnet` and `if_proto` structures in relation to a generic protocol and a generic interface.

Figure 1-6 Sample `ifnet` structure in relation to a protocol and a network driver



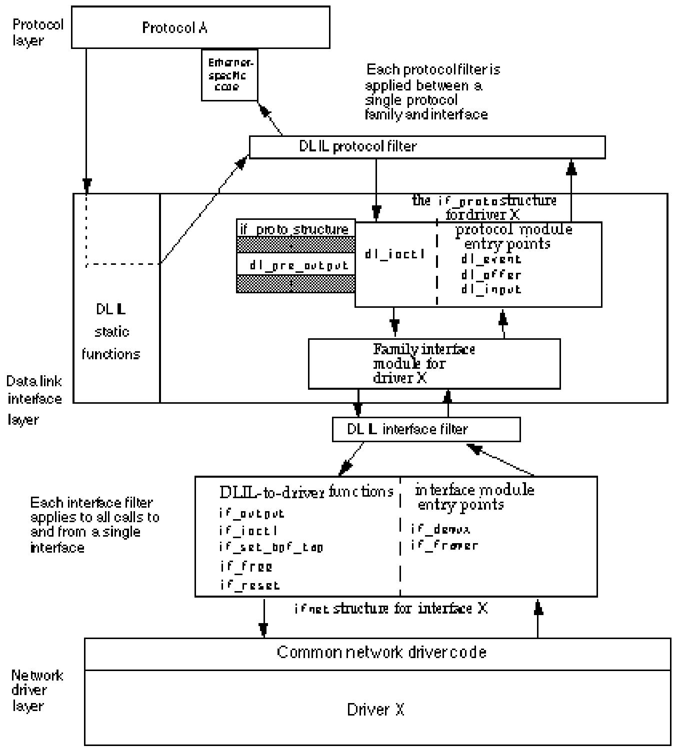
Installing and Removing Data Link NKEs

To support the dynamic insertion of filters into the data and control streams between the network layer and the interface layer and the removal of inserted filters, the DLIL defines the following static functions:

- `dlil_attach_protocol_filter`, which inserts an NKE between the DLIL and one of the attached protocols. Such an extension is known as a DLIL protocol filter. This type of NKE provides access to all function calls between the DLIL and the attached protocol for a specific protocol/interface pair.
- `dlil_attach_interface_filter`, which inserts an NKE between the DLIL and an attached interface. Such a filter is known as an DLIL interface filter. This type of NKE provides access to all frames flowing to or from an interface.
- `dlil_detach_filter`, which removes previously inserted DLIL protocol and interface filters.

Figure 1-7 (page 21) shows the relationship of protocol and interface filters to the protocol stack layer, DLIL, and network driver layer.

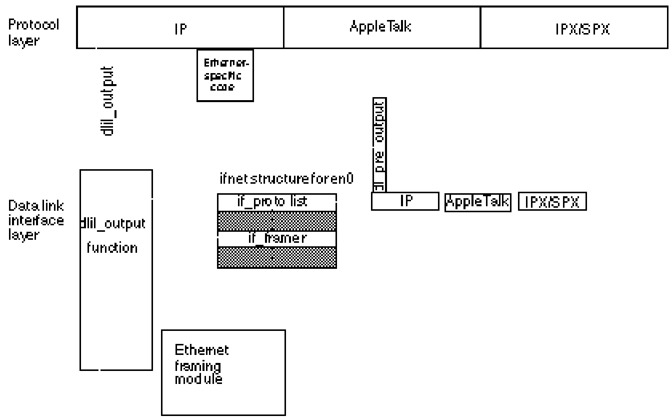
Figure 1-7 Protocol and interface extensions in relation to the DLIL



Sending Data

Figure 1-8 (page 22) shows the sequence of calls required to send an IP packet over the MACE Ethernet interface (`en0`).

Figure 1-8 Example of sending an IP packet



The following steps correspond to the numbers in Figure 1-8 (page 22) and describe the process of sending a packet:

1. The `ip_output` routine in the IP protocol stack calls `dlil_output`, passing the `dl_tag` value for the stack's attachment to `en0`.

2. Using the `dl_tag` value, the `dlil_output` function locates the `dl_pre_output` pointer in the `if_proto` structure for IP.

3. The `dlil_output` function uses the `dl_pre_output` pointer in the `if_proto` structure to call IP's interface-specific output module. This module calls its `arpresolve` routine to resolve the target IP address into a media access control (MAC) address.

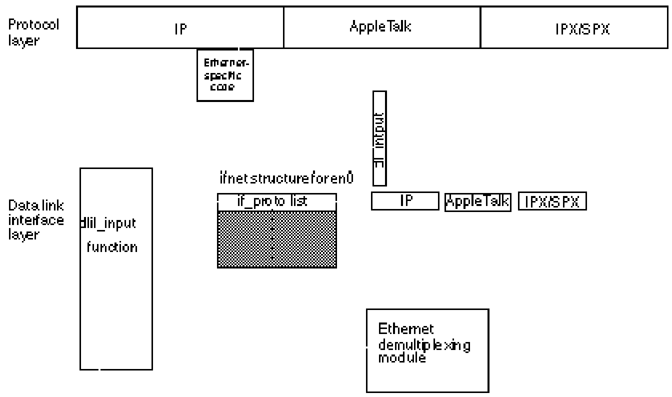
4. When IP's interface-specific output module returns, the `dlil_output` function uses the `if_framer` pointer in the `ifnet` structure to call the appropriate framing function in the DLIL interface module. The framing function prepends interface-specific frame data to the packet.

5. The `dlil_output` function calls the function pointed to by the `if_output` field in the `ifnet` structure for `en0` and sends the frame to the MACE Ethernet driver.

Receiving Data

Figure 1-9 (page 23) shows the sequence of calls required to receive an IP packet from the MACE Ethernet interface (`en0`).

Figure 1-9 Example of receiving a packet



The following steps correspond to the numbers in Figure 1-9 (page 23) and describe the process of receiving a packet:

1. The MACE Ethernet driver or its support code calls `dlil_input` with pointers to its `ifnet` structure and `mbuf` chain.

2. The `dlil_input` function uses the `if_demux` entry in the `ifnet` structure to call the demultiplexing function for the interface family (Ethernet in this case).

3. The demultiplexing function identifies the frame and returns an `if_proto` pointer to `dlil_input`.

4. The `dlil_input` function calls the protocol input module through the `dl_input` pointer in the `if_proto` structure.

Note: The Ethernet-specific module for IP receives the frame, removes the 802.2 or SNAP header (if any) and delivers the packet to the protocol's `ipintr` routine.

For more information

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

- *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/

Using Network Kernel Extensions

This chapter provides an overview for the TCPLogger sample which is included in the NKE documentation package.

> **Important:** The information provided in this document is relevant for Mac OS 10.1 through 10.3. Specific mention is made for items which apply to newer releases of the OS.
>
> For all shipping releases of Mac OS X prior to 10.4, the Network Kernel Extensions (NKE) APIs have not been officially supported. The legacy NKE architecture was implemented as an interim solution. The legacy API was never designed to be officially supported. Other aspects of the OS X networking implementation have received a higher priority, and so the interim solution has remained in effect to OS X 10.3.x.
>
> The NKE mechanism for Mac OS X version 10.4 and later is described in the document *Network Kernel Extensions Programming Guide*.

Example: TCPLogger

tcplognke is a socket NKE which is invoked for each TCP connection. 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 command line utility demonstrates control of the tcplognke NKE to enable/disable logging, dump log information, and specify different logging criteria.

When tcplognke is loaded and initialized, it installs itself in the TCP protocol structure ready for use and it registers a Kernel Controller structure. The tcplog utility demonstrates the use of the PF_SYSTEM socket to enable/disable logging in the tcplognke, to have the NKE send saved log information to the tool, for the tool to display in the terminal window. Other command options are implemented in the tool to control the operations of the NKE.

The tcplognke NKE 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 NKE gathers, use the enclosed tcplog command line utility. The tool configures the tcplognke NKE to send log records to the tcplog program. The tcplog tool then loops, displaying and writing log records as the tcplognke NKE creates them.

The source code for the tcplognke NKE and for the tcplog command line utility are available for the *current* (10.4 and later) version of the NKE architecture as the *tcplognke* sample code project. See the Read Me file with the TCPLogger sample code for more instructions on the design and use of the sample NKE.

The *legacy* tcplognke NKE (for 10.3 and earlier) is not published and is not supported. You must contact Apple developer technical support to obtain this sample code.

Network Kernel Extensions Reference

Important: The information provided in this document is relevant for Mac OS 10.1 through 10.3. Specific mention is made for items which apply to newer releases of the OS.

For all shipping releases of Mac OS X prior to 10.4, the Network Kernel Extensions (NKE) APIs have not been officially supported. The legacy NKE architecture was implemented as an interim solution. The legacy API was never designed to be officially supported. Other aspects of the OS X networking implementation have received a higher priority, and so the interim solution has remained in effect to OS X 10.3.x.

The NKE mechanism for Mac OS X version 10.4 and later is described in the document *Network Kernel Extensions Programming Guide*.

This chapter describes the functions that NKEs can call and NKE-specific data types. The functions are organized into the following sections:

- "Kernel Utilities" (page 27) lists the kernel utilities that NKEs can call.
- "protosw Functions" (page 31) describes functions that access the `protosw` structure.
- "ifaddr Functions" (page 32) describes functions that access the `ifnet` structure.
- "mbuf Functions" (page 33) describes functions that access the `mbuf` structure.
- "Socket Functions" (page 34) describes functions that access the `socket` structure.
- "Socket Buffer Functions" (page 36) describes functions that access the sockbuf structure.
- "Protocol Family NKE Functions" (page 40) describes NKE functions that protocol families call.
- "Protocol Handler NKE Functions" (page 41) describes NKE functions that protocol handlers call.
- "Data Link NKE Functions" (page 42) describes functions that data link NKEs call.
- "NKE Structures and Data Types" (page 60) describes the NKE structures and data types.

Kernel Utilities

NKEs can call the following kernel utility functions:

- `_MALLOC` (page 28)
- `_FREE` (page 28)
- `kalloc` (page 28)
- `kfree` (page 29)
- `kprintf` (page 29)
- `psignal` (page 29)

_MALLOC

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.These 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

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 _MALLOC

kalloc

Allocate kernel memory.

```
void *kalloc(vm_size_t size);
```

This is roughly the kernel equivalent of `malloc`. (See usage note at kfree (page 29).) This should generally be used for memory associated with the loading and unloading of an NKE. For storing data coming from outside sources (such as an `mbuf`), `_MALLOC` is more appropriate.

kfree

Frees memory allocated with `kalloc` (page 28).

```
void kfree(void *data, vm_size_t size);
```

This behaves much like the user-space `free`.

> **Note:** The kernel allocator does not keep track of allocation size. You are responsible for passing in the original size yourself. If you pass in a different size, unpredictable behavior may result.

kprintf

Print text to the console.

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

Identical to `printf` in user space. It is not safe to call `kprintf` 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.

psignal

Sends a signal to a user process.

```
void psignal(struct proc *p, int sig);
```

splimp

Sets priority level to prevent execution of any kernel thread whose priority is less than or equal to "IMP".

```
spl_t splimp(void);
```

In effect, this prevents any concurrency with anything that would touch the networking stack's data structures at any level. The function returns the previous spl level in a form suitable to pass to `splx` (page 30).

Note: IMP is a legacy term, short for Interface Message Processor, which was essentially the early term for a network interface card (NIC).

splnet

Sets priority level to prevent execution of any kernel thread whose priority is less than or equal to "Net".

```
spl_t splnet(void);
```

This blocks interrupts from network devices and any execution that would result from those interrupts (at the network stack level, not the I/O Kit level). The function returns the previous spl level in a form suitable to pass to `splx` (page 30).

splx

Restores a previously-saved priority level.

```
void splx(spl_t level);
```

The value passed in generally is the result of the use of another spl function. The `spl_t` type is an unsigned int.

suser

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

```
int suser(struct proc *proc);
```

timeout

Sets a timeout for the next call to `tsleep` (page 31).

```
void timeout(void (*)(void *)call_on_timeout, void *arg, int ticks);
```

The function `call_on_timeout` is called after some number of ticks. This timeout can be removed with `untimeout` (page 31).

The length of a tick is system dependent, but the number of ticks per second can be obtained from the global variable `HZ`.

This function returns immediately, and is usually followed by some operation, followed by a `tsleep` (page 31) call while waiting for the operation to complete. The operation (occurring in some other thread) would typically end by calling `untimeout` (page 31) to prevent the error handler from being triggered, followed by a call to `wakeup` (page 31) to actually wake the thread.

tsleep

Sleep until an event is posted with `wakeup` (page 31) or until a timeout occurs. This is commonly combined with a timeout value to bound the wait.

```
int tsleep(void *chan, int pri, const char *wmesg, int timo);
```

The timeout value is measured in ticks. The length of a tick is system-dependent, but the number of ticks per second can be obtained from the global variable `HZ`. To sleep until woken (as one might reasonably do when used in conjunction with a call to `timeout` (page 30), you should pass the value zero (0) for `timo`.

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. After another thread has called `wakeup` (page 31) 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.

This is frequently used in conjunction with `timeout` (page 30) and `untimeout` (page 31).

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.

untimeout

Unregisters a timeout previously registered with `timeout` (page 30).

```
void untimeout(void (*)(void *), void *arg);
```

Note that `untimeout` does not wake the thread that called `timeout` (page 30). If this is desired, you must explicitly do so using `wakeup` (page 31).

wakeup

Wakes a thread that is sleeping through a call to `tsleep` (page 31).

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

protosw Functions

This section describes the functions that access the `protosw` structure.

pffindproto

The `pffindproto` function obtains the `protosw` corresponding to the protocol family, protocol, and protocol type (or `NULL`). These values are passed to the `socket`(2) call from user mode.

```
extern struct protosw *pffindproto(int, int, int);
```

pffindtype

The `pffindtype` function obtains the `protosw` corresponding to the protocol and protocol type requested. These values are passed to the `socket`(2) call from user mode.

```
extern struct protosw *pffindtype(int, int);
```

ifaddr Functions

This section describes the functions that access the `ifaddr` structure.

ifa_ifwithaddr

The `ifa_ifwithaddr` function searches the `ifnet` list for an interface with a matching address.

```
struct ifaddr *ifa_ifwithaddr(struct sockaddr *);
```

ifa_ifwithdstaddr

The `ifa_ifwithdstaddr` function searches the `ifnet` list for an interface with a matching destination address.

```
struct ifaddr *ifa_ifwithdstaddr(struct sockaddr *);
```

ifa_ifwithnet

The `ifa_ifwithnet` function searches the `ifnet` list for an interface with the most specific matching address.

```
struct ifaddr *ifa_ifwithnet(struct sockaddr *);
```

ifa_ifwithaf

The `ifa_ifwithaf` function searches the `ifnet` list for an interface with the first matching address family.

```
struct ifaddr *ifa_ifwithaf(int);
```

ifa_ifafree

The `ifa_ifafree` function frees the specified `ifaddr` structure.

```
void ifafree(struct ifaddr*);
```

ifa_ifaof_ifpforaddr

The ifa_ifaof_ifpforaddr function searches the address list in the ifnet structure for the one matching the sockaddr structure. The matching rules are exact match, destination address on point-to-point link, matching network number, or same address family.

```
struct ifaddr *ifaof_ifpforaddr(struct sockaddr *, struct ifnet *);
```

mbuf Functions

For a description of the mbuf routines, go to FreeBSD website (http://www.freebsd.org), click on the Manual-pages reference and search for "mbuf".

```
struct mbuf *m_copy(struct mbuf *, int, int, int);
struct mbuf *m_free(struct mbuf *);
struct mbuf *m_get(int, int);
struct mbuf *m_getclr(int, int);
struct mbuf *m_gethdr(int, int);
struct mbuf *m_prepend(struct mbuf *, int, int);
struct mbuf *m_pullup(struct mbuf *, int);
struct mbuf *m_retryhdr(int, int);
void m_adj(struct mbuf *, int);
int m_clalloc(int, int);
void m_freem(struct mbuf *);
struct mbuf *m_devget(char *, int, int, struct ifnet, void );
void m_cat(struct mbuf *, struct mbuf *);
void m_copydata(struct mbuf *, int, int, caddr_t);
void m_freem(struct mbuf *);
int m_leadingspace(struct mbuf *);
int m_trailingspace(struct mbuf *);
```

Caution About Using Malloc'd Memory In mbufs

Prior to the release of the Power Mac G5 and to Mac OS X 10.3.x, it was possible to use malloc'd memory to hold data or packet headers on outbound mbufs. For compatibility with all G5's and Mac OS X 10.3.x and later, any memory, containing outbound packet data or headers, must have a recognized I/O Address compatible with PCI Address Translation.

When there is a requirement to obtain memory buffers to contain outbound packet data or headers, use the standard mbuf routines as defined in </sys/mbuf.h> to allocate mbufs for your NKE needs. To understand more about this requirement, refer to Tech Note 2090 "Driver Tuning on Panther or G5" Section "Background Info on PCI Address Translation".

The standard mbuf routines have been modified to provide memory buffers, which are compatible for use with the PCI bus. If your driver allocates memory with malloc, fills packet data into this memory, points the mbuf to this memory, and sends the mbuf to the driver to be sent across the network, one symptom may be that the hardware hangs on trying to send the packet.

Note that the NKE may continue to `malloc`/`free` memory for it's own needs, such as for internal processing of packet data. However, when processing is complete, outbound packet data must be placed in a buffer with a recognized I/O Address, compatible with PCI Address Translation.

An additional note about memory allocation/de-allocation - `malloc`/`free` are potential preemption points and may loose the funnel, which means that the NKE can get re-entered while doing a blocking malloc or a free. While a free call is not likely to result in the loss of a funnel, it can happen. When doing a `malloc`/`free`, the NKE should make sure that it's a safe point for the integrity of the data structures that it manipulates. For more information about funnels, refer to the following web references:

- http://www.kernelthread.com/mac/osx/arch_xnu.html
- http://www.usenix.org/publications/library/proceedings/bsdcon02/full_papers/gerbarg/gerbarg_html/

Socket Functions

This section describes the socket functions. For additional information on the use of these functions, read *TCP/IP Illustrated, Volume 2 - The Implementation* by Gary Wright and W. Richard Stevens, Addison Wesley, ISBN 0-201-63354-X.

soabort

The `soabort` function calls the protocol's `pr_abort` function at `slpnet`.

```
soabort(struct socket *);
```

soaccept

The `soaccept` function calls the protocol's `pr_accept` function.

```
soaccept(struct socket *, struct mbuf *);
```

sobind

The `sobind` function calls the protocol's `pr_bind` function.

```
sobind(struct socket *, struct mbuf *);
```

soclose

The `soclose` function aborts pending and in-progress connections, calls `sodisconnect` for connected sockets, and sleeps if any connections linger or block. It then calls the protocol's `pr_detach` function and frees the socket.

```
soclose(struct socket *);
```

soconnect

If connected or connecting, the `soconnect` function tries to disconnect. It also calls the `pr_connect` function.

```
soconnect(struct socket *, struct mbuf *);
```

soconnect2

The `soconnect2` function calls the `pr_connect2` function. This function is generally not supported, but it is used to support pipe usage in the `AF_LOCAL` domain.

```
soconnect2(struct socket *, struct socket *);
```

socreate

The `socreate` function links the `protosw` structure and the socket. It calls the protocol's `pr_attach` function.

```
socreate(int, struct socket**, int, int);
```

sodisconnect

The `sodisconect` function calls the protocol's `pr_disconnect` function.

```
sodisconnect(struct socket *);
```

sofree

The `sofree` function removes the caller from `q0` and `q` queues, releases the send `sockbuf`, flushes the receive `sockbuf`, and frees the socket.

```
sofree(struct socket *);
```

sogetopt

The `sogetopt` function processes `SOL_SOCKET` requests and always calls the `PRCO_SETOPT` function.

```
sogetopt(struct socket *, int, int, struct mbuf **);
```

sohasoutofband

The `sohasoutofband` function indicates that the caller has an out-of-band notifier.

```
sooutofband(struct socket *);
```

solisten

The `solisten` function calls the protocol's `pr_listen` function and sets the queue backlog.

```
solisten(struct socket *, int);
```

soreceive

The `soreceive` function receives data.

```
soreceive(struct socket *, struct mbuf **, struct uio *, struct mbuf **, struct
 mbuf **, int *);
```

soflush

The `soflush` function locks the socket, marks it as "can't receive," unlocks the socket, and calls `sbrelease`.

```
soflush(struct socket *);
```

sosend

The `sosend` function sends data.

```
sosend(struct socket *, struct mbuf *, struct uio *, struct mbuf *, struct mbuf
 *, int);
```

sosetopt

The `sosetopt` function processes `SOL_SOCKET` requests and always calls the `PRCO_SETOPT` function.

```
sosetopt(struct socket *, int, int, struct mbuf *);
```

soshutdown

The `soshutdown` function calls the `sorflush function` (`FREAD`) and the `pr_shutdown` function (`FWRITE`).

```
soshutdown(struct socket *, int, int, struct mbuf *);
```

Socket Buffer Functions

This section describes the socket buffer functions.

sb_lock

The `sb_lock` function locks a `sockbuf` structure. It sets `WANT` and sleeps if the structure is already locked.

```
sb_lock(struct sockbuf *);
```

sbappend

The `sbappend` function conditionally calls `sbappendrecord` and calls `sbcompress`.

```
sbappend(struct sockbuf *, struct mbuf *);
```

sbappendaddr

The `sbappendaddr` function conditionally calls `sbappendrecord` and `sbcompress`.

```
sbappendaddr(struct sockbuf *, struct sockaddr *, struct mbuf *, struct mbuf
*);
```

sbappendcontrol

The `sbappendcontrol` function calls `sbspace` and `sballoc`.

```
sbappendcontrol(struct sockbuf *, struct mbuf *, struct mbuf *);
```

sbappendrecord

The `sbappendrecord` function calls `sballoc` and `sbcompress`.

```
sbappendrecord(struct sockbuf *, struct mbuf *);
```

sbcompress

The `sbcompress` function calls `sballoc`.

```
sbcompress(struct sockbuf *, struct mbuf *, struct mbuf *);
```

sbdrop

The `sbdrop` function calls `sbfree`.

```
sbdrop(struct sockbuf *, int);
```

sbdroprecord

The `sbdroprecord` function calls `sbfree`.

```
sbdroprecord(struct sockbuf *);
```

sbflush

The `sbflush` function calls `sbfree`.

```
sbflush(struct sockbuf *);
```

sbinsertoob

The `sbinsertoob` function calls `sballoc` and `sbcompress`.

```
sbinsertoob(struct sockbuf *, struct mbuf *);
```

sbrelease

The `sbrelease` function calls `sbflush` and clears the `selwait` structure.

```
sbrelease(struct sockbuf *);
```

sbreserve

The `sbreserve` function sets up the `sockbuf` counts.

```
sbreserve(struct sockbuf *, u_long);
```

sbwait

The `sbwait` function sets `SB_WAIT and` calls `tsleep` on `sb_cc`.

```
sbwait(struct sockbuf *);
```

socantrcvmore

The `socantrcvmore` function marks socket and wakes up readers.

```
socantrcvmore(struct socket *);
```

socantsendmore

The `socantsendmore` function marks socket and wakes up writers.

```
socantsendmore(struct socket *);
```

soisconnected

The `soisconnected` function sets state bits. It calls `soqremque`, `soqinsque`, `sorwakeup`, and `sowwakeup`.

```
soisconnected(struct socket *);
```

soisconnecting

The `soisconnecting` function sets state bits.

```
soisconnecting(struct socket *);
```

soisdisconnected

The `soisdisconnected` function sets state bits, calls timer wakeup, and wakes up readers and writers.

```
soisdisconnected(struct socket *);
```

soisdisconnecting

The `soisdisconnecting` function sets state bits, calls timer wakeup, and wakes up readers and writers.

```
soisdisconnecting(struct socket *);
```

su_sonewconn1

The `su_sonewconn1` function allocates socket, sets state, inserts into head queue, and calls `pr_attach`.

```
struct socket *su_sonewconn1(struct socket *, int);
```

soqinsque

The `soqinsque` function adds the socket to `q` or `q0` of "head."

```
soqinsque(struct socket *, struct socket *, int);
```

soqremque

The `soqremque` function removes socket from `q` or `q0` of "head."

```
soqremque(struct socket *, int);
```

soreserve

The `soreserve` function sets up send and receive `sockbuf` structures.

```
soreserve(struct socket *, u_long, u_long);
```

Protocol Family NKE Functions

This section describes the functions that support the dynamic addition and removal of protocol family NKEs. For additional descriptions of these routines, go to the FreeBSD website, click on the Manual-pages reference and search for "net_add_domain".

net_add_domain

Adds a `domain` structure to the kernel's domain list.

```
void net_add_domain(struct domain *domain);
```

`domain`On input, a pointer to a `domain` structure to be linked into the system's list of domains.

`function result:` None.

DISCUSSION

The `net_add_domain` function adds a `domain` (represented by the `domain` parameter) to the kernel's list of domains.

The `net_add_domain` function locks the `domain` structure, calls the domain's `init` function, and calls the protocol's `init` function for each attached protocol. The domain's `init` function updates certain system global structures, such as `max_protohdr`, and protects itself from repeated calls. You can choose whether to include the `protosw` structures in `domain`. The alternative is to attach protocol handler NKEs by calling "net_add_proto" (page 41).

This function does not return a value because it cannot fail.

net_del_domain

Removes a `domain` structure from the kernel's domain list.

```
int net_del_domain(struct domain *domain);
```

`domain`On input, a pointer to the `domain` structure that is to be removed.

`function result`0 to indicate success, `EBUSY` when the reference count for the specified `domain` structure is not zero, and `EPFNOSUPPORT` if the specified `domain` structure cannot be found.

DISCUSSION

The `net_del_domain` function removes a `domain` structure from the kernel's list of `domain` structures.

You are responsible for reclaiming resources and handling dangling pointers before you call `net_del_domain`.

This function is only called from a domain implementation.

pffinddomain

Finds a `domain`.

```
struct domain *pffinddomain(int x);
```

`x`On input, a `PK` constant, such as `PF_INET` or `PF_NKE`.

`function result`A pointer to the requested `domain` structure or `NULL`, which indicates that the domain could not be found. If `pffinddomain` returns `NULL`, the caller should return `EPFNOSUPPORT` in addition to performing normal error cleanup.

DISCUSSION

The `pffinddomain` function locates the `domain` structure for the specified protocol family in the kernel's list of `domain` structures.

Note: This function depends on matching an integer value with a value in the kernel. You can verify that the proper `domain` structure has been located by checking the value of the `dom_name` field in the `domain` structure.

Protocol Handler NKE Functions

This section describes the functions that support the dynamic addition and removal of protocol handler NKEs.

net_add_proto

Adds the specified `protosw` structure to the list of `protosw` structures for the specified `domain`.

```
int net_add_proto (struct protosw *protosw, struct domain *domain);
```

`protosw`On input, a pointer to a `protosw` structure.

`domain`On input, a pointer to a `domain` structure.

`function result`0 to indicate success or `EEXISTS` if the `pr_type` and the `pr_protocol` fields in the `protosw` structure that is being added match the `pr_type` and `pr_protocol` fields in an existing `protosw` entry for the specified domain.

DISCUSSION

The `net_add_proto` function adds the specified `protosw` to the domain's list of `protosw` structures.

If the `protosw` structure is successfully added, the protocol's `init` function (if present) is called.

net_del_proto

Removes a `protosw` structure from the list of `protosw` structures for the specified `domain`.

```
int net_del_proto(int type, int protocol, struct domain *domain);
```

`type:` On input, an integer value that specifies the type of the `protosw` structure that is to be removed.

`protocol:` On input, an integer value that specifies the protocol of the `protosw` structure that is to be removed.

`domain:` On input, a pointer to a `domain` structure.

`function result:` 0 to indicate success or `ENXIO` if the specified values for `type` and `protocol` don't match a `protosw` structure in the domain's list of `protosw` structures.

DISCUSSION

The `net_del_proto` function removes the specified `protosw` structure from the list of `protosw` structures for the specified `domain` structure.

Data Link NKE Functions

This section describes the Data Link Layer Interface (DLIL) functions. The section is organized under the following topics:

- "Calling the DLIL From the Network Layer" (page 43)
- "Calling the Network Layer From the DLIL" (page 48)
- "Calling the Driver Layer From the DLIL" (page 50)
- "Calling the DLIL From the Driver Layer" (page 52)
- "Calling Interface Modules From the DLIL" (page 56)
- "Calling the DLIL From a DLIL Filter" (page 58)

Calling the DLIL From the Network Layer

This section describes DLIL functions that are called from the network layer. The functions are

- "dlil_attach_protocol_filter" (page 43) which is called to attach a protocol filter.
- "dlil_attach_interface_filter" (page 44) which is called to attach an interface filter.
- "dlil_attach_protocol" (page 45) which a protocol calls to attach itself to the DLIL.
- "dlil_detach_filter" (page 45) which a protocol calls to attach itself to the DLIL.
- "dlil_detach_protocol" (page 46) which a protocol calls to deattach itself from the DLIL.
- "dlil_output" (page 46) which a protocol calls to send data to a network interface.
- "dlil_ioctl" (page 47) which a protocol calls to send ioctl commands to a network interface.

dlil_attach_protocol_filter

Inserts a DLIL protocol filter between a protocol and the DLIL.

```
int dlil_attach_protocol_filter( u_long dl_tag, struct dlil_pr_flt_str
*protocol_filter, u_long *filter_id, int insertion_point);
```

`dl_tag`On input, a value of type `u_long`, previously obtained by calling "dlil_attach_protocol" (page 45), that identifies the protocol/interface pair between which the NKE is to be inserted.

`protocol_filter`A pointer to a `dlil_pr_fil_str` structure that contains pointers to the functions the DLIL is to call when it intercepts calls. Each function pointed to by a member of this structure corresponds to a function pointed to by the `ifnet` structure for this protocol/interface pair.

`filter_id`On input, a pointer to a `u_long`. On output, `filter_id` points to a tag value that identifies the NKE that has been inserted. The tag value is required to remove the NKE or insert another NKE after the current NKE.

`insertion_point`On input, a value of type `int`. If this is the first DLIL protocol filter to be inserted, set `insertion_point` to `DLIL_LAST_FILTER`. If this is the second or greater insertion, set `insertion_point` to the value of `filter_id` returned by a previous call to `dlil_attach_protocol_filter` or to `DLIL_LAST_FILTER` to insert the filter at the end of the chain of inserted filters.

`function result`0 for success.

DISCUSSION

The `dlil_attach_protocol_filter` function inserts a DLIL protocol filter between the specified protocol and the DLIL.

When more than one DLIL protocol filter is inserted, the DLIL calls the appropriate function of the first filter with the parameters provided by the caller. When that call returns successfully, the DLIL calls the appropriate function for the second filter with the parameters returned by the first filter, and so on until the appropriate functions have been called for each filter in the list. When the last filter in the list has been called, the DLIL calls the original destination function with the parameters returned by the last filter.

The DLIL skips any function pointers that are `NULL`, which allows DLIL protocol filters to intercept only a subset of the calls that may be made by a protocol to the interface to which the protocol is attached.

If a DLIL protocol filter returns a status other zero (which indicates success) or EJUSTRETURN, the DLIL frees any associated mbuf chain (for the filter_dl_pre_output and filter_dl_input functions only) and returns with that status.

If a DLIL protocol filter returns a status of EJUSTRETURN, the DLIL returns zero to indicate success without freeing any associated mbuf chain. The DLIL protocol filter is responsible for freeing or forwarding any associated mbuf chain.

dlil_attach_interface_filter

Inserts a DLIL interface filter between the DLIL and the interface.

```
int dlil_attach_interface_filter( struct ifnet *ifnet_ptr,
        struct dlil_if_flt_str *interface_filter,
        int *filter_id,
        u_long insertion_point);
```

ifnet_ptrA pointer to the ifnet structure for this interface.

interface_filterA pointer to a dlil_if_fil_str structure that contains pointers to the function calls that the DLIL is to call when the family interface module calls common network driver code for the specified interface. Each function pointed to by a member of this structure corresponds to a function pointed to by the ifnet structure.

filter_idOn input, a pointer to a value of type int. On output, filter_id points to a value that identifies the NKE that has been inserted. This value is required to remove the NKE or insert another NKE after it.

insertion_pointOn input, a value of type u_long. If this is the first insertion, set insertion_point to DLIL_LAST_FILTER. If this is the second or greater insertion, set insertion_point to the value of filter_id returned by a previous call to dlil_attach_interface_filter or to DLIL_LAST_FILTER to insert the filter at the end of the chain of inserted filters.

function result0 for success. Other possible errors are defined in <errno.h>.

DISCUSSION

The dlil_attach_interface_filter function inserts a DLIL interface filter between the DLIL and an interface. When the filter is in place, the DLIL intercepts all calls between itself and the interface's driver and passes the call and its parameters to the filter.

You can insert multiple DLIL interface filters, in which case the DLIL calls the filters in the order specified by insertion_point at the time of insertion. The order in which filters are executed is reversed when an incoming packet is being processed (that is, the last filter called for an outbound packet will be the first filter called for an inbound packet).

When more than one DLIL interface filter is installed, the DLIL calls the appropriate function for the first filter with the parameters provided by the caller. When that call returns successfully, the DLIL calls the appropriate function for the second filter with the parameters returned by the first filter, and so on until the appropriate functions have been called for each filter in the list. When the last filter has been called, the DLIL calls the original destination function with the parameters returned by the last filter.

The DLIL skips any null function pointers, which allows DLIL interface filters to intercept only a subset of the calls that the DLIL may make to the driver for the specified interface.

If a DLIL interface filter returns a status other than zero (which indicates success) or EJUSTRETURN, the DLIL frees any associated mbuf chain (for the filter_if_output and filter_if_input functions only) and returns with that status.

If a DLIL interface extension returns a status of EJUSTRETURN, the DLIL returns zero to indicate success. The DLIL interface filter is responsible for freeing or forwarding any associated mbuf chain.

With a return value of zero, the DLIL continues to process the list of NKEs.

dlil_attach_protocol

Attaches a protocol to the DLIL for use with an interface.

```
int dlil_attach_protocol( struct dlil_proto_reg_str *proto_reg, u_long *dl_tag);
```

proto_regOn input, a pointer to a "dlil_proto_reg_str" (page 61) structure containing all of the information required to complete the attachment.

dl_tagOn input, a pointer to a value of type u_long. On output, dl_tag points to an opaque value identifying the interface/protocol pair that is passed in subsequent calls to the dlil_output, dlil_ioctl, and dlil_detach functions.

function result0 for success and ENOENT if the specified interface does not exist. Other possible errors are defined in <errno.h>.

DISCUSSION

The dlil_attach_protocol function attaches a protocol to the DLIL for use with a specific network interface. For example, you would call dlil_attach_protocol to attach the TCP/IP protocol family to en0, which is the first Ethernet family interface.

dlil_detach_filter

Removes a DLIL interface filter or a DLIL protocol filter.

```
int dlil_detach_filter( u_long filter_id );
```

filter_idA value of type u_long obtained by previously calling "dlil_attach_interface_filter" (page 44) or "dlil_attach_protocol_filter" (page 43).

function result0 for success or ENOENT of the specified filter does not exist.

DISCUSSION

The dlil_detach_filter function removes a DLIL interface filter or a DLIL protocol filter that was previously attached by calling "dlil_attach_interface_filter" (page 44) or "dlil_attach_protocol_filter" (page 43).

If the filter has a detach routine and a function pointer to it was supplied when the filter was attached, the DLIL calls the filter's detach routine before detaching the filter. The detach routine should complete any clean up tasks before it returns.

dlil_detach_protocol

Detaches a protocol from the DLIL.

```
int dlil_detach_protocol( u_long dl_tag );
```

`dl_tag`On input, a value of type `u_long`, previously obtained by calling "dlil_attach_protocol" (page 45), that identifies the protocol and the interface from which the protocol is to be detached.

`function result`0 for success and `ENOENT` if the defined protocol is not currently attached. Other possible errors are defined in `<errno.h>`.

DISCUSSION

The `dlil_detach_protocol` function detaches a protocol that was previously attached to the DLIL by calling "dlil_attach_protocol" (page 45). Before detaching the protocol, the DLIL calls the detach filter callback functions for any NKEs that may have been inserted between the protocol and the interface that is being detached from.

The DLIL keeps a reference count of protocols attached to each interface. When the reference count reaches zero as a result of calling `dlil_detach_protocol`, the DLIL calls the "if_free" (page 52) function for the affected interface to notify the driver that no protocols are attached to the interface. The reference count can only reach zero if the driver detaches the interface.

dlil_output

Sends data to a network interface.

```
int dlil_output (u_long dl_tag, struct mbuf *buffer, caddr_t route, struct
sockaddr *dest, int raw);
```

`dl_tag`On input, a value of type `u_long`, previously obtained by calling "dlil_attach_protocol" (page 45), that identifies the associated protocol/interface pair.

`buffer`On input, a pointer to the `mbuf` chain, which may contain multiple packets.

`route`On input, a pointer to an opaque pointer-sized value whose use is specific to each protocol family, or `NULL`.

`dest`On input, a pointer to an `sockaddr` structure that defines the target network address that the DLIL passes to the associated `dl_pre_output` function. If `raw` is `FALSE`, this parameter is ignored.

`raw`On input, a Boolean value. Setting `raw` to `TRUE` indicates that the `mbuf` chain pointed to by `buffer` contains a link-level frame header (which means that no further processing by the protocol or by the interface family modules is required). If `raw` is `FALSE`, protocol filters are not called, but any interface filters attached to the target interface are called.

`function result`0 for success.

DISCUSSION

The `dlil_output` function is a DLIL function that the network layer calls in order to send data to a network interface. The `dlil_output` function executes as follows:

1. If the raw parameter is TRUE, go to step 4. Otherwise, if the raw parameter is FALSE and the attached protocol identified by dl_tag has defined a dl_pre_output function, the DLIL calls that dl_pre_output function and passes to it all of the parameters passed to dl_output by the caller, as well as pointers to two buffers in which the dl_pre_output function can pass back the frame type and destination data link address.

2. If any data link protocol extensions are attached to the protocol/interface pair, those NKEs are called in the order they were inserted. If any NKE returns a value other than zero for success or EJUSTRETURN, the DLIL stops processing the packet, dlil_output frees the mbuf chain, and returns an error to its caller. When any NKE returns EJUSTRETURN, packet processing terminates without freeing the mbuf chain. In this case, the NKE is responsible for freeing or forwarding the mbuf chain.

3. If an if_framer function is defined for this interface, the DLIL calls the if_framer function. The if_framer function adds any necessary link-level framing to the outbound packet. This function usually prepends the frame header to the beginning of the mbuf chain.

4. If any data link interface NKEs have been attached to the interface specified by dl_tag, those NKEs are called in the order they were inserted. If any NKE returns a value other than zero for success or EJUSTRETURN, the DLIL stops processing the packet, frees the mbuf chain, and returns an error to its caller. When any NKE returns EJUSTRETURN, packet processing terminates without freeing the mbuf chain. In this case, the NKE is responsible for freeing or forwarding the mbuf chain.

5. As the last step, dlil_output calls if_output in order to pass the mbuf chain and a pointer to the ifnet structure to the interface's driver.

dlil_ioctl

Accesses DLIL-specific or driver-specific functionality.

```
int dlil_ioctl (u_long dl_tag, struct ifnet *ifp, u_long ioctl_code, caddr_t
ioctl_arg);
```

dl_tagOn input, a value of type u_long, previously obtained by calling “dlil_attach_protocol” (page 45), that identifies the associated protocol/interface pair. If not zero, the DLIL uses the value of dl_tag to identify the target protocol module. If dl_tag is zero, ifp is not NULL, and the interface has defined an if_ioctl function, the DLIL calls the interface's if_ioctl function and passes to it the parameters supplied by the caller.

ifpOn input, a pointer to the ifnet structure associated with the target interface. This parameter is not used if dl_tag is non-zero.

ioctl_codeOn input, a value of type u_long that specifies the specific ioctl function that is to be accessed.

ioctl_argOn input, a value of type caddr_t whose contents depend on the value of ioctl_code.

function result0 for success.

DISCUSSION

The dlil_ioctl function is a DLIL function that the network layer calls in order to send ioctl commands to a network interface.

Calling the Network Layer From the DLIL

This section describes network layerfunctions called by the DLIL. The functions are

- "dl_pre_output" (page 48), which the DLIL calls in order to perform protocol-specific processing (such as resolving the network address to a link-level address) for outbound packets.
- "dl_input" (page 49), which the DLIL calls in order to pass incoming packets to the protocol.
- "dl_offer" (page 49), which the DLIL calls in order to identify incoming frames.
- "dl_event" (page 50), which the DLIL calls in order to pass events from the driver layer to a protocol.

dl_pre_output

Obtains the destination link address and frame type for outgoing packets.

```
int (*dl_pre_output) (struct mbuf *mbuf_ptr, caddr_t route_entry, struct sockaddr
 *dest, char *frame_type, char *dest_linkaddr, u_char dl_tag);
```

`mbuf_ptr`On input, a pointer to an `mbuf` structure containing one or more outgoing packets.

`route_entry`On input, a value of type `caddr_t` that is passed to the DLIL when a protocol calls "dlil_output" (page 46).

`dest`On input, a pointer to a `sockaddr` structure that describes the packets' destination network address, or `NULL`. This parameter is passed to the DLIL when the protocol calls "dlil_output" (page 46). The format of the `sockaddr` structure is specific to each protocol family.

`frame_type`On input, a pointer to a byte array of undefined length. On output, `frame_type` contains the frame type for this protocol.

`dest_linkaddr`On input, a pointer to a byte array of undefined length. On output, `dest_linkaddr` contains the destination link address.

`dl_tag`On input, a value of type `u_long`, previously obtained by calling "dlil_attach_protocol" (page 45), that identifies the associated protocol/interface pair.

`function result`0 for success. Errors are defined in `<errno.h>`.

DISCUSSION

The `dl_pre_output` function obtains the link address and frame type for outgoing packets whose destination is described by the `dest` parameter.

The `dl_pre_output` function pointer in the `if_proto` structure is optionally defined when a protocol calls the function "dlil_attach_protocol" (page 45) to register a protocol family. The DLIL calls the `dl_pre_output` function when a protocol calls "dlil_output" (page 46).

In addition to defining the destination link address and the frame type, the `dl_pre_output` function may also add a packet header, such as 802.2 or SNAP.

dl_input

Receives incoming packets.

```
int (*dl_input) (struct mbuf *mbuf_ptr, char *frame_header, struct ifnet
*ifnet_ptr, caddr_t dl_tag, int sync_ok);
```

`mbuf_ptr`On input, a pointer to an `mbuf` structure.

`frame_header`On input, a pointer to a byte array of undefined length containing the frame header.

`ifnet_ptr`On input, a pointer to the `ifnet` structure for this protocol/interface pair.

`dl_tag`On input, a value of type `u_long`, previously obtained by calling "dlil_attach_protocol" (page 45), that identifies the associated protocol/interface pair.

`sync_ok`Reserved.

`function result`0 for success. Errors are defined in `<errno.h>`.

DISCUSSION

The `dl_input` function is called by the DLIL. When a DLIL module receives a frame from the driver and finishes interface-specific processing, it calls the target protocol through the `dl_input` function pointer. The interface family's demultiplexing module identifies the target protocol by matching the data provided in the demultiplexing descriptors when the protocol was attached.

The `dl_input` function pointer in the `if_proto` structure is defined by the `input` member of the "dlil_proto_reg_str" (page 61) structure, which the function "dlil_attach_protocol" (page 45) passes to the DLIL when a protocol is attached.

dl_offer

Examines unidentified frames.

```
int (*dl_offer) (struct mbuf *mbuf_ptr, char *frame_header; u_long dl_tag);
```

`mbuf_ptr`On input, a pointer to an `mbuf` structure containing incoming frames.

`dl_tag`On input, a value of type `u_long`, previously obtained by calling "dlil_attach_protocol" (page 45), that identifies the associated protocol/interface pair.

`frame_header`On input, a pointer to a byte array containing the frame header as received from the driver. The length of `frame_header` depends on the interface family.

`function result``DLIL_FRAME_ACCEPTED` or `DLIL_FRAME_REJECTED`.

DISCUSSION

The `dl_offer` function accepts or rejects a frame that was not identified by a protocol's demultiplexing descriptors.

When the interface family demultiplexing module receives a frame that does not match any of the protocol's demultiplexing descriptors, the module calls any defined dl_offer function and passes to it the unidentified frame. The dl_offer function can accept or reject the frame.

The dl_offer function pointer in the if_proto structure is optionally defined by the offer member of the "dlil_proto_reg_str" (page 61) structure, which the "dlil_attach_protocol" (page 45) function passes to the DLIL when a protocol is attached.

If a dl_offer function accepts the frame, the frame is not offered to any other protocol's dl_offer function. If no dl_offer function accepts the frame, the frame is dropped.

Note: The dl_offer function only indicates whether it will accept the frame. It does not modify the frame or start processing it. Processing occurs when dlil_input calls the protocol's dl_input function.

dl_event

Receives events passed by the DLIL from the interface's driver.

```
void (*dl_event) (struct event_msg *event, u_long dl_tag);
```

eventOn input, a pointer to an event_msg structure.

dl_tagOn input, a value of type u_long, previously obtained by calling "dlil_attach_protocol" (page 45), that identifies the associated protocol/interface pair. The dl_event function uses dl_tag to determine the interface that was the source of the event.

function resultNone.

DISCUSSION

The dl_event function receives events from the interface's driver. When the DLIL receives an event from the driver, the module calls the defined dl_event functions of all protocols that are attached to the interface, passing in event_msg an event-specific code and an event value that is interpreted by the dl_event function.

If "dlil_attach_protocol" (page 45) was called with a null pointer for the dl_event function, no action is taken for that protocol family.

The dl_event function pointer in the if_proto structure is optionally defined by the event member of "dlil_proto_reg_str" (page 61) structure, which "dlil_attach_protocol" (page 45) passes to the DLIL when a protocol is attached.

Calling the Driver Layer From the DLIL

The functions described in this section are called by the DLIL to an interface's driver. The functions are

- "if_output" (page 51), which the DLIL calls in order to pass outgoing packets to the interface's driver.
- "if_ioctl" (page 51), which the DLIL calls in order to pass ioctl commands to the interface's driver.
- "if_set_bpf_tap" (page 51), which the DLIL calls in order to enable or disable a binary packet filter tap.
- "if_free" (page 52), which the DLIL calls in order to free the ifnet structure for an interface.

if_output

Accepts outgoing packets and passes them to the interface's driver.

```
int (*if_output) (struct ifnet *ifnet_ptr, struct mbuf *buffer);
```

`ifnet_ptr`On input, a pointer to the `ifnet` structure for this interface.

`buffer`On input, a pointer to an `mbuf` structure containing one or more outgoing packets.

`function result`0 for success. Errors are defined in `<errno.h>`.

DISCUSSION

The `if_output` function sends outgoing packets to the interface's driver. The DLIL calls `if_output` when the associated protocol calls "dlil_output" (page 46).

The `if_output` function must accept all of the packets in the `mbuf` chain.

The `if_output` function pointer is defined in the interface's `ifnet` structure and is initialized by the interface driver before the interface driver calls "dlil_if_attach" (page 53).

if_ioctl

Processes ioctl commands.

```
int (*if_ioctl) (struct ifnet *ifnet_ptr, u_long ioctl_code, caddr_t ioctl_arg);
```

`ifnet_ptr`On input, a pointer to the `ifnet` structure for this interface.

`ioctl_code`On input, a value of type `u_long` containing the ioctl command.

`ioctl_arg`On input, a value of type `caddr_t` whose contents depend on the value of `ioctl_code`.

`function result`0 for success. Other results are specific to the driver's ioctl function.

DISCUSSION

The `if_ioctl` function accepts and processes ioctl commands that access driver-specific functionality.

The `if_ioctl` pointer is defined in the interface's `ifnet` structure and is initialized by the interface driver before the interface driver calls "dlil_if_attach" (page 53).

if_set_bpf_tap

Enables or disables a binary packet filter tap for an interface.

```
int (*if_set_bpf_tap) (int mode, struct ifnet *ifnet_ptr, void (*bpf_callback)
 ( struct ifnet *ifnet_ptr, struct mbuf *mbuf_ptr, int direction);
```

`mode`On input, a value of type `int` that is `BPF_TAP_DISABLE` (to disable the tap), `BPF_TAP_INPUT` (to enable the tap on incoming packets), `BPF_TAP_OUTPUT` (to enable the tap on outgoing packets), or `BPF_TAP_INPUT_OUTPUT` (to enable the tap on incoming and outgoing packets).

`ifnet_ptr`On input, a pointer to the `ifnet` structure for this interface.

`callback`On input, a function pointer to the tap.

`function result`0 for success.

DISCUSSION

The `if_set_bpf_tap` function enables or disables a read-only binary packet filter tap for an interface. A tap is different from a NKE in that it is read-only and that it operates within the driver. Any network driver attached to the DLIL can be tapped.

The `if_set_bpf_tap` function pointer is defined in the interface's `ifnet` structure by the driver before the driver calls "dlil_if_attach" (page 53).

If the value of the `mode` parameter is `BPF_TAP_INPUT`, `BPF_TAP_OUTPUT`, or `BPF_TAP_INPUT_OUTPUT`, the `bfp_callback` parameter points to a C function the driver calls when transmitting or receiving data over the interface (depending on the value of `mode`). If the value of `mode` is `BPF_TAP_DISABLE`, the tap is disabled for incoming and outgoing packets.

When the driver calls its `bpf_callback` function, it passes a pointer to the interface's `ifnet` structure and a pointer to the incoming or outgoing `mbuf` chain.

if_free

Frees the `inet` structure for an interface.

```
void (*if_free) (struct ifnet *ifnet_ptr);
```

`ifnet_ptr`On input, a pointer to the `ifnet` structure that is to be freed.

`function result`None.

DISCUSSION

The `if_free` function frees the `ifnet` structure for an interface. It is called by the DLIL in response to a previous `dlil_if_detach` call from the driver that returned `DLIL_WAIT_FOR_FREE`. Once all references to the `ifnet` structure have been deallocated, the DLIL calls "if_free" (page 52) to notify the driver that the associated `ifnet` structure pointed to by `ifnet_ptr` is no longer being referenced and can be deallocated.

The `if_free` pointer is defined in the interface's `ifnet` structure before the interface driver calls "dlil_if_attach" (page 53).

Calling the DLIL From the Driver Layer

Drivers call the following DLIL functions:

- "dlil_if_attach" (page 53) to attach an interface to the DLIL.
- "dlil_if_detach" (page 53) to detach an interface from the DLIL.
- "dlil_reg_if_modules" (page 54) to register an interface family module.

- "dlil_find_dl_tag" (page 54) to locate the `dl_tag` value for a protocol and interface family pair.
- "dlil_input" (page 55) to pass incoming packets to the DLIL.
- "dlil_event" (page 55) to pass event codes to the DLIL.

dlil_if_attach

Attaches an interface to the DLIL for use by a specified protocol.

```
int dlil_if_attach( struct ifnet *ifnet_ptr );
```

`ifnet_ptr`A pointer to an `ifnet` structure containing all of the information required to complete the attachment. The `ifnet` structure may be embedded within an interface-family-specific structure, in which case the `ifnet` structure must be the first member of that structure.

`function result`0 for success and `ENOENT` if no interface family module is found. Other possible errors are defined in `errno.h`.

DISCUSSION

The `dlil_if_attach` function attaches an interface to the DLIL. If the DLIL interface family module for the specified interface has not been loaded, an error is returned. (See "dlil_reg_if_modules" (page 54).)

The DLIL calls the "add_if" (page 56) function for the interface family module in order to initialize the module's portion of the `ifnet` structure and perform any module-specific tasks. At minimum, the `add_if` function is responsible for initializing the "if_demux" (page 57) and `if_framer` function pointers in the `ifnet` structure. Later, the DLIL uses the `if_demux` function pointer to call the demultiplexing descriptors for the interface in order to demultiplex incoming frames and uses the `if_framer` function pointer to frame outbound packets.

Once `add_if` initializes the members of the `ifnet` structure for which it is responsible, the DLIL places the interface on the list of network interfaces, and `dlil_if_attach` returns.

dlil_if_detach

Detaches an interface from the DLIL.

```
int dlil_if_detach( struct ifnet *ifnet_ptr );
```

`inet_ptr`A pointer to an `ifnet` structure that was previously used to call "dlil_if_attach" (page 53).

`function result`0 for success. `DLIL_WAIT_FOR_FREE` if the driver must wait for the DLIL to call the "if_free" (page 52) callback function before deallocating the `ifnet` structure.

DISCUSSION

The `dlil_if_detach` function detaches a network interface from the DLIL, thereby disabling communication to and from the interface. Then the DLIL marks the interface as detached in the interface's `ifnet` structure. To notify the protocols that are attached to the interface that the interface has been detached, the DLIL then calls the `dl_event` function for all of the protocols have defined such a function. In response, attached protocols should call `dlil_detach_protocol` to detach themselves from the interface.

The protocols or the socket layer may still have references to the ifnet structure for the detached interface, so interface drivers should wait to deallocate the interface's ifnet structure until the DLIL calls the interface's "if_free" (page 52) function to notify the driver that all protocols have detached from the interface.

dlil_reg_if_modules

Registers an interface family.

```
dlil_reg_if_modules(u_longinterface_family, int (*add_if), int (*del_if), int
(*add_proto), int (*del_proto), int (*shutdown)());
```

interface_familyOn input, a value of type u_long specified that uniquely identifies the interface family. Values for the current interface families are defined in <net/if_var.h>. You can define new interface family values by contacting DTS.

add_ifOn input, a pointer to the interface family module's add_if function.

del_ifOn input, a pointer to the interface family module's del_if function.

add_protoOn input, a pointer to the interface family module's add_proto function.

del_protoOn input, a pointer to the interface family module's del_proto function.

shutdownOn input, a pointer to the interface family module's shutdown function.

function result0 for success. Other errors are defined in errno.h.

DISCUSSION

The dlil_reg_if_modules function registers an interface family module that contains the necessary functions for processing inbound and outbound packets including if_demux and if_framer functions. Any null function pointers are skipped in DLIL processing.

dlil_find_dl_tag

Gets the dl_tag for an interface and protocol family pair.

```
dlil_find_dl_tag(u_longif_family; short unit; u_long proto_family; u_long
*dl_tag);
```

if_familyOn input, a value of type u_long that uniquely identifies the interface family. See <net/if_var.h> for possible values.

unitOn input, a value of type short containing the unit number of the interface.

proto_familyOn input, a value of type u_long that uniquely identifies the protocol family. See <net/if_var.h> for possible values.

dl_tagOn input, a pointer to a value of type u_long in which the dl_tag value for the specified interface and protocol family pair is to be returned.

function result0 for success. EPROTONOSUPPORT if a matching pair is not found.

DISCUSSION

The `dlil_find_dl_tag` function locates the `dl_tag` value associated with the specified interface and protocol family pair.

dlil_input

Passes incoming packets to the DLIL.

```
int dlil_input(struct ifnet *ifp, struct mbuf *m);
```

`ifp`On input, a pointer to the `ifnet` structure for this interface.

`m`On input, a pointer to the head of a chain of `mbuf` structures containing one or more incoming frames.

`function result`0 for success.

DISCUSSION

The `dlil_input` function is called by the driver layer to pass incoming frames from an interface to the DLIL. The `dlil_input` function performs the following sequence:

1. Any interface filters attached to the associated interface are called.
2. Assuming all filters return successfully, "if_demux" (page 57) is called to determine the target protocol family. If `if_demux` cannot find a matching protocol family, `dlil_input` calls the `dl_offer` functions (if any) defined by the attached protocol families.
3. If no target protocol family is found, the frame is dropped.
4. Any protocol filters attached to the target protocol family/interface are called.
5. If all protocol filters return successfully, the frame is passed to the protocol family's `dl_input` function. DLIL frame processing is finished.

dlil_event

Notifies the DLIL of significant events.

```
void (*dlil_event) (struct ifnet *ifnet_ptr, struct event_msg *event);
```

`ifnet_ptr`On input, a pointer to the `ifnet` structure for this interface.

`event`On input, a pointer to an `event_mgs` structure containing a unique event code and a pointer to event data.

`function result`A result code.

DISCUSSION

The dlil_event function is called by the driver layer to pass event codes, such as a change in the status of power management, to the DLIL. The DLIL passes a pointer to the ifnet structure for this interface and the event parameter to those protocols that are attached to this interface and that have provided a pointer to a dl_event function for receiving events. The protocols may or may not react to any particular event code.

Calling Interface Modules From the DLIL

The DLIL calls the following interface module functions:

- "add_if" (page 56) to add an interface.
- "del_if" (page 56) to remove an interface.
- "add_proto" (page 57) which is called to add a protocol.
- "del_proto" (page 57) which is called to remove a protocol.

add_if

Adds an interface.

```
int (*add_if) struct ifnet *ifp);
```

ifpOn input, a pointer to the ifnet structure for the interface that is being added.

function result0 for success.

DISCUSSION

The add_if function is called by the DLIL in response to a call to "dlil_if_attach" (page 53). The DLIL calls add_if in the interface family module in order to initialize the module's portion of the ifnet structure and perform any module-specific tasks.

At minimum, the add_if function initializes the "if_demux" (page 57) and if_framer function pointers in the ifnet structure. Later, the DLIL uses the if_demux function pointer to call the demultiplexing function for the interface to demultiplex incoming frames and calls the if_framer function to frame outbound packets.

del_if

Deinitializes portions of an ifnet structure.

```
int (*del_if) struct ifnet *ifp);
```

ifpOn input, a pointer to the ifnet structure for the interface that is being deinitialized.

function result0 for success.

DISCUSSION

The del_if function is called by the DLIL to notify an interface family module that an interface is being detached. The interface family module should remove any references to the interface and associated structures.

add_proto

Adds a protocol.

```
int (*add_proto)(struct ddesc_head_str *demux_desc_head) struct if_proto *proto,
 u_long dl_tag);
```

demux_desc_headOn input, a pointer to the head of a linked list of one or more protocol demultiplexing descriptors for the protocol that is being added.

protoOn input, a pointer to the if_proto structure for the protocol that is being added.

dl_tagOn input, a value of type u_long, previously obtained by calling "dlil_attach_protocol" (page 45), that identifies the associated protocol/interface pair.

function result0 for success.

DISCUSSION

The add_proto function is an interface family module function that processes the passed demux descriptor list, extracting any information needed to identify the attaching protocol in subsequent incoming frames.

del_proto

Removes a protocol.

```
int (*del_proto) (struct if_proto *proto, u_long dl_tag);
```

protoOn input, a pointer to the if_proto structure for the protocol that is being removed.

dl_tagOn input, a value of type u_long, previously obtained by calling "dlil_attach_protocol" (page 45), that identifies the associated protocol/interface pair.

function result0 for success.

DISCUSSION

The del_proto function is called by the DLIL to remove a protocol family from an interface family module's list of attached protocol families. Any references to the associated if_proto structure pointer should be removed before returning.

if_demux

Locates demultiplexing descriptors.

```
void (*if_demux) (struct ifnet *ifnet_ptr, struct mbuf *mbuf_ptr, char *
frame_header);
```

`ifnet_ptr`On input, a pointer to the `ifnet` structure for this interface.

`mbuf_ptr`On input, a pointer to an `mbuf` structure containing one or more incoming frames.

`frame_header`On input, a pointer to a character string in the `mbuf` structure containing a frame header.

`function result`0 for success.

DISCUSSION

The `if_demux` function is an interface family function called by "dl_input" (page 49) to determine the target protocol family for an incoming frame. This function uses the demultiplexting data passed in from previous calls to the `add_proto` function. When a match is found, `if_demux` returns the associated `if_proto` pointer.

Calling the DLIL From a DLIL Filter

DLIL filters call the following DLIL functions in order to inject data into a data path:

- "dlil_inject_if_input" (page 58) is called by a DLIL interface filter to inject frames into the inbound data path.
- "dlil_inject_if_output" (page 59) is called by a DLIL interface filter to inject packets into the outbound data path.
- "dlil_inject_pr_input" (page 59) is called by a DLIL protocol filter to inject frames into the inbound data path.
- "dlil_inject_pr_output" (page 60) is called by a DLIL protocol filter to inject packets into the output data path.

dlil_inject_if_input

Injects frames into the inbound data path from the interface filter level.

```
int dlil_inject_if_input (struct mbuf *buffer, char *frame_header, ulong from_id);
```

`buffer`On input, a pointer to a chain of `mbuf` structures containing the packets that are to be injected.

`frame_header`On input, a pointer to a byte array of undefined length containing the frame header for the frames that are to be injected.

`from_id`On input, a value of type `ulong` containing the filter ID of the calling filter obtained by previously calling "dlil_attach_interface_filter" (page 44). If `from_id` is set to `DLIL_NULL_FILTER`, all attached interface filters are called.

`function result`0 for success.

DISCUSSION

The `dlil_inject_if_input` function is called by an interface filter NKE to inject frames into the inbound data path. The frames can be frames that the filter generates or frames that were previously consumed.

When a filter injects a frame, the DLIL invokes all of the input interface filter NKEs that would normally be invoked after the filter identified by `filter_id`. The behavior is identical to the processing of a frame passed to "dlil_input" (page 55) from the driver layer except that all interface filter NKEs preceding and including the injecting filter are not executed.

dlil_inject_if_output

Injects packets into the outbound data path from the interface filter level.

```
int dlil_inject_if_output ( struct mbuf *buffer, ulong from_id);
```

`buffer`On input, a pointer to a chain of `mbuf` structures containing the packets that is to be injected.

`from_id`On input, a value of type `ulong` containing the filter ID of the calling filter obtained by previously calling "dlil_attach_interface_filter" (page 44). If `from_id` is set to `DLIL_NULL_FILTER`, all attached interface filters are called.

`function result`0 for success.

DISCUSSION

The `dlil_inject_if_output` function is called by an interface filter NKE to inject frames into the outbound data path. The packets can be packets that the filter generates or packets that were previously consumed.

When a filter injects a packet, the DLIL invokes all of the output interface filter NKEs that would normally be invoked after the filter that calls "dlil_inject_if_output" (page 59). This behavior is identical to the last steps of packet processing done by `dlil_output`, except that all output interface filter NKEs preceding and including the injecting filter are not executed.

Note: The injected packets must contain any frame header that the driver layer requires.

dlil_inject_pr_input

Injects frames into the inbound data path from the protocol filter level.

```
int dlil_inject_pr_input (struct mbuf *buffer, char *frame_header, ulong from_id);
```

`buffer`On input, a pointer to a chain of `mbuf` structures containing the data that is to be injected.

`frame_header`On input, a pointer to a byte array of undefined length containing the frame header for the frames that are to be injected.

`from_id`On input, the filter ID of the calling filter obtained by previously calling "dlil_attach_protocol_filter" (page 43). If `from_id` is set to the constant `DLIL_NULL_FILTER`, all attached interface filters are called.

`function result`0 for success.

DISCUSSION

The `dlil_inject_pr_output` function is called by a protocol filter NKE to inject frames into the outbound data path. The frames can be frames that the filter generates or frames that were previously consumed.

When a protocol filter calls dlil_inject_pr_output, the DLIL invokes all of the input protocol filter NKEs that would normally be invoked after the filter that calls dlil_inject_pr_input. This behavior is identical to the last steps of processing that occur when a frame is passed to "dl_input" (page 49), except that all protocol filter NKEs preceding and including the injecting filter are not executed.

dlil_inject_pr_output

Injects packets into the outbound data path from the protocol filter level.

```
int dlil_inject_pr_output (struct mbuf *buffer,
        struct sockaddr *dest, int raw, char *frame_type,
        char *dst_linkaddr, ulong from_id);
```

bufferOn input, a pointer to a chain of mbuf structures containing the data that is to be injected.

destOn input, a pointer to an opaque pointer-sized variable whose use is specific to each protocol family, or NULL.

rawOn input, a Boolean value. Setting raw to TRUE indicates that the mbuf chain pointed to by buffer contains a link-level frame header, which means that no further processing by the protocol or the interface family modules is required. The value of raw does not affect whether the DLIL calls any NKEs that are attached to the protocol/interface pair.

frame_typeOn input, a pointer to a byte array of undefined length containing the frame type. The length and content of frame_type are specific to each interface family.

dst_linkaddrA pointer to a byte array of undefined length containing the destination link address.

from_idOn input, a value of type ulong containing the filter ID of the calling filter obtained by previously calling "dlil_attach_protocol_filter" (page 43). If from_id is set to DLIL_NULL_FILTER, all attached interface filters are called.

function result0 for success.

DISCUSSION

The dlil_inject_pr_output function is called by a protocol filter NKE to inject packets into the outbound data path. The packets can be packets that the filter generates or packets that were previously consumed.

When a protocol filter calls dlil_inject_pr_output, the DLIL invokes all of the output protocol filter NKEs that would normally be invoked after the filter that calls dlil_inject_pr_output. This behavior is identical to the execution of dlil_output following the call to dl_pre_output except that all output protocol filters preceding and including the injecting filter are not executed.

NKE Structures and Data Types

This section describes the NKE structures and data types. The structures are

- "dlil_proto_reg_str" (page 61) which provides the information necessary to attach a protocol to the DLIL.

- "dlil_demux_desc" (page 62) which provides the information necessary to identify a protocol's packets.
- "dlil_if_flt_str" (page 63) which contains pointers to all of the functions the DLIL may call when sending or receiving a frame from an interface.
- "dlil_pr_flt_str" (page 65) which contains pointers to all of the functions the DLIL may call when it passes a call to an NKE.

Note: With the exception of the `ifnet` structure, the DLIL makes its own copy of all structures that are passed to it.

dlil_proto_reg_str

The `dlil_proto_reg_str` structure is passed as a parameter to the "dlil_attach_protocol" (page 45) function, which attaches network protocol stacks to interfaces.

```
struct dlil_proto_reg_str {
    struct ddesc_head_str demux_desc_head;
    u_long interface_family;
    u_long protocol_family;
    short unit_number;
    int default_proto;
    dl_input_func input;
    dl_pre_output_func pre_output;
    dl_event_func event;
    dl_offer_func offer;
    dl_ioctl_func ioctl;
};
```

Field Descriptions

`ddesc_head_str` The head of a linked list of one or more protocol demultiplexing descriptors. Each demultiplexing descriptor defines several sub-structures that are used to identity and demultiplex incoming frames belonging to one or more attached protocols. When multiple methods of frame identification are used for an interface family, a chain of demultiplexing descriptors may be passed to "dlil_attach_protocol" (page 45) and to "add_if" (page 56) to identify each method.

`interface_family`A unique unsigned long value that specifies the interface family. Values for current interface families are defined in `<net/if_var.h>`. Developers may define new interface family values through DTS.

`protocol_family`A unique unsigned long value defined that specifies the protocol family being attached. Values for current protocol families are defined in `<net/dlil.h>`. Developers may define new protocol family values through DTS.

`unit_number`Specifies the unit number of the interface to which the protocol is to be attached. Together, the `interface_family` and `unit_number` fields identify the interface to which the protocol is to be attached.

`default_proto`Reserved. Always 0.

input Contains a pointer to the function that the DLIL is to call in order to pass input packets to the protocol stack.

pre_outputContains a pointer to the function that the DLIL is to call in order to perform protocol-specific processing for outbound packets, such as adding an 802.2/SNAP header and defining the target address.

eventContains a pointer to the function that the DLIL is to call in order to notify the protocol stack of asynchronous events, or is NULL. If this field is NULL, events are not passed to the protocol stack.

offer Contains a pointer to the function that the DLIL is to call in order to offer a frame to the attached protocol, or is NULL. If offer is NULL, the DLIL will not be able to offer frames that cannot be identified to the protocol and the frame may be dropped.

ioctlContains a pointer to the function that the DLIL is to call in order to send ioctl calls to the interface's driver.

dlil_demux_desc

The dlil_demux_desc structure is a member of the "dlil_proto_reg_str" (page 61) structure. The dlil_demux_desc structure is the head of a linked list of protocol demultiplexing descriptors that identify the protocol's packets in incoming frames.

```
struct dlil_demux_desc {
    TAILQ_ENTRY(dlil_demux_desc) next;
    int type;
    u_char *native_type;
    union {
        struct {
            u_long proto_id_length;
            u_char *proto_d;
            u_char *proto_id_mask;
        } bitmask;
        struct {
            u_char dsap;
            u_char ssap;
            u_char control_code;
            u_char pad;
        } desc_802_2;
        struct {
            u_char dsap;
            u_char ssap;
            u_char control_code;
            u_char org[3];
            u_short protocol_type;
        } desc_802_2_SNAP;
    } variants;
} TAILQ_HEAD {
    ddesc_head_str,
    dlil_demux_desc
};
```

Field Descriptions

nextA link pointer used to chain multiple descriptors.

typeSpecifies which variant of the descriptor has been defined. For Ethernet, the possible values are DESC_802_2, DESC_802_2_SNAP, and DESC_BITMASK.

native_typeA pointer to a byte array containing a self-identifying frame ID, such as the two-byte Ethertype field in an Ethernet II frame. This field may be used by itself, may be used in combination with other identifying information, or may not be used at all, in which case, its value is NULL.

variantsThree structures that comprise a union. The bitmask structure describes any combination of bits that identify frames that do not match Ethernet 802.2 frames and Ethernet 802.2/SNAP frames. The desc_802_2 structure and the desc_802_2_SNAP structure describe Ethernet 802.2 frames and Ethernet 802.2/SNAP frames, respectively.

For each Ethernet interface, the following sequence must take place. The actual implementation may optimize the process.

1. The first if_proto structure is referenced. The structure is found through the proto_head pointer in the associated ifnet structure.

2. The frame is compared with the first demultiplexing descriptor in the protocol's list of demultiplexing descriptors (the bitmask structure).

3. If the native type is NULL or if the interface family's frame doesn't have a frame type field, go to step 4. Otherwise, the octet string in native-type is compared with the interface family's native frame-type specification field. The frame format for each interface family defines the number of bits to compare. If there is a match and the proto_id and proto_id_mask fields are defined, go to step 4. If there is a match and the proto_id and proto_id_mask fields are NULL, the frame is passed to the protocol's input function, thereby terminating DLIL processing of the frame.

4. If the proto_id or proto_id_mask fields in the bitmask structure are NULL, or if the proto_id_length field is 0, go to step 5. Otherwise, compare the first proto_id_length bytes of the frame's data field with proto_id, ignoring any bits defined as zero in the proto_id_mask. If there is a match, the frame is passed to the protocol's input function, thereby terminating DLIL processing of the frame.

5. This demultiplexing descriptor could not provide a match. Advance to the next demultiplexing descriptor in the list and go to step 3.

6. None of the demultiplexing descriptors could provide a match. If there is another if_proto structure in the interface's protocol list, go back to step 2 using the first demultiplexing descriptor for this protocol.

7. No match could be found using any demultiplexing descriptor for any of the protocols attached to the interface. Go back through the if_proto structures for the attached protocols and call any defined dl_offer function. If a dl_offer function returns DLIL_FRAME_ACCEPTED, the DLIL passes the frame to the responding protocol's dl_input function, thereby terminating DLIL processing of the frame.

8. None of the protocols attached to this interface have accepted the frame. The mbuf chain is freed and the frame is dropped.

The bitmask structure or one of the predefined 802.2 structures can be used to identify frames.

dlil_if_flt_str

The dlil_ir_flt_str structure is a parameter to the "dlil_attach_interface_filter" (page 44) function, which inserts DLIL interface filters between the DLIL and an interface.

This structure contains pointers to all of the functions that are called at the point at which the filter is placed.

```
struct dlil_if_flt_str {
        caddr_t cookie;
        int (*filter_if_input)(caddr_t cookie,
            struct ifnet **ifnet_ptr,
            struct mbuf **mbuf_ptr,
            char **frame_ptr);
        int (*filter_if_event)(caddr_t cookie,
            struct ifnet **ifnet_ptr,
            struct event_msg **event_msg_ptr);
        int (*filter_if_output)(caddr_t cookie,
            struct ifnet **ifnet_ptr,
            struct mbuf **mbuf_ptr);
        int (*filter_if_ioctl)(caddr_t cookie,
            struct ifnet **ifnet_ptr,
            u_long ioctl_code_ptr,
            caddr_t ioctl_arg_ptr);
        int (*filter_if_free)(caddr_t cookie,
            struct ifnet **ifnet_ptr);
        int (*filter_detatch)(caddr_t cookie);
```

Field Descriptions

`filter_if_input`A pointer to the `filter_if_input` function for this DLIL interface filter. The parameters for this function are `cookie`, (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments), a pointer to the `ifnet` structure for this interface, a pointer to an `mbuf` structure, and pointer to the frame.

`filter_if_event`A pointer to the `filter_if_event` function for this DLIL interface filter. The parameters for this function are `cookie`, (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments), a pointer to the `ifnet` structure for this interface, and a pointer to an `event_msg` structure containing the event that is being passed to the extension.

`filter_if_output`A pointer to the `filter_if_output` function for this DLIL interface filter. The parameters for this function are `cookie` (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments), a pointer to the `ifnet` structure for this interface, and a pointer to the memory buffer for this packet.

`filter_if_ioctl`A pointer to the `filter_if_ioctl` function for this DLIL interface filter. The parameters for this function are `cookie` (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments), a pointer to the `ifnet` structure for this interface, an unsigned long that points to the I/O control code for this call, and a pointer to parameters that the DLIL passes to the `filter_if_ioctl` function.

`filter_if_free`A pointer to the `filter_if_free` function for this DLIL interface filter. The parameters for this function are `cookie` (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments) and a pointer to the `ifnet` structure for this interface.

`filter_detach`A pointer to the `filter_detach` function for this DLIL interface filter. The parameter for this function is `cookie` (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments). For details, see "dlil_detach_filter" (page 45).

dlil_pr_flt_str

The dlil_pr_flt_str structure is a parameter to the function "dlil_attach_protocol_filter" (page 43), which inserts DLIL protocol filters between a protocol and the DLIL.

This structure contains pointers to all of the functions that are called at the point at which the filter is placed.

```
struct dlil_if_flt_str {
    caddr_t cookie;
    int (*filter_dl_input)(caddr_t cookie,
            struct mbuf **m,
            char **frame_header,
            struct ifnet **ifp);
    int (*filter_dl_output)(caddr_t cookie,
            struct mbuf **m,
            struct ifnet **ifp,
            struct sockaddr **dest,
            char *dest_linkaddr,
            char *frame_type);
    int (*filter_dl_event)(caddr_t cookie,
            struct event_msg *event_msg);
    int (*filter_dl_ioctl)(caddr_t cookie,
            struct ifnet **ifp,
            u_long ioctl_cmd,
            caddr_t ioctl_arg);
    int (*filter_detach)(caddr_t cookie);
};
```

Field Descriptions

filter_dl_inputA pointer to the filter_dl_input function for this DLIL protocol filter. The parameters for this function are cookie, (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments), a pointer to an mbuf structure, and a pointer to the ifnet structure for the interface.

filter_dl_outputA pointer to a filter_dl_output function for this DLIL protocol filter. The parameters for this function are cookie (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments), a pointer to the ifnet structure for the interface, a pointer to the socket address for this destination, a pointer to the link address for this destination, and a pointer to the frame type.

filter_dl_eventA pointer to the filter_pr_event function for this DLIL protocol filter. The parameters for this function are cookie, (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments), a pointer to the ifnet structure for the interface, and a pointer to an event_msg structure containing the event that is being passed to the extension.

filter_dl_ioctlA pointer to a filter_if_ioctl function for this DLIL protocol filter. The parameters for this function are cookie (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments), a pointer to the ifnet structure for the interface, an u_long that points to the I/O control command for this call, and a pointer to parameters that the DLIL passes to the filter_if_ioctl function.

`filter_detach`A pointer to the `filter_detach` function for this DLIL protocol filter. The parameter for this function is `cookie` (an opaque value that is passed by the filter and is returned so that the filter can identify one attachment among many attachments).

Document Revision History

This table describes the changes to *Network Kernel Extensions (legacy)*.

Date	Notes
2006-10-03	Clarified the availability of sample code.
2005-10-04	Corrected some minor formatting issues.
2005-08-11	Fixed error in code sample.
2005-06-04	Fixed minor typographical errors.
2005-04-29	Updated title to reflect legacy status. (This document covers Mac OS X v10.3 and earlier.)
2004-04-22	Initial republication

Glossary

domain A complete protocol family.

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

Data Link Interface Layer (DLIL) The fixed part of the network kernel extension architecture that exists between protocol stacks and the network drivers.

data link interface module A network kernel extension that handles demultiplexing or packet framing.

data link NKE A network kernel extension that exists between the protocol stacks and the device layer.

DLIL interface filter A network kernel extension that is installed between the DLIL and one or more network interfaces.

DLIL protocol filter A network kernel extension that is installed between the DLIL and a network protocol stack.

data link protocol module A network kernel extension that handles the specific interface for the protocol's attachment to a particular interface family.

global NKE An NKE that is automatically enabled for sockets of the type specified for the NKE.

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.

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

programmatic filter NKE An NKE that is enabled only under program control, using socket options, for a specific socket.

protocol family NKE A network kernel extension that implements a domain.

protocol handler A network kernel extension that implements a specific protocol within a domain.

socket NKE A network kernel extension that is installed between the socket layer and the protocol stack or network device layers.