

# LibPPF Primer

Cipherica Labs  
May 2003

## Overview

Libppf is Packet Processing Framework for Windows OS geared towards abstracting network packet interception and manipulation code from the specifics of the host platform.

## Features

*Libppf* creates NDIS-independent packet processing environment by implementing a set of methods, which:

- intercept media-level network traffic on physical, virtual and WAN adapters
- dynamically track the state and the presence of these adapters
- and provide ancillary IP layer services

The core feature of *libppf* is simplicity of its API.

*It's everything you need from the network stack, nothing you don't.*

By providing a complete per-adapter information and enabling unrestricted packet inspection and manipulation the library delivers a solid foundation for a variety of network applications including:

- traffic firewalling, routing and masquerading
- traffic shaping
- load balancing
- VPNs and tunnelling
- implementations of custom layer 2 protocols
- and arbitrary real-time traffic reengineering

Per-adapter information includes:

- a system adapter ID
- a descriptive name
- a media information - a protocol, an address, an MTU and a link speed

Per-packet information indicates an interface and the direction the packet was travelling in.

The library also implements a set of IP layer services, which significantly simplify handling of tunnelled and encapsulated traffic. This part of the library deals with:

- monitoring IP address assignments to the adapters
- monitoring changes made to the routing table
- finding an interface and the next hop for a given remote IP address

The library also comes in a special version, which emulates the exact kernel environment in the userspace. This enables writing and debugging the code with conventional development tools and then seamlessly moving it into the kernel for a production release.

## Structure

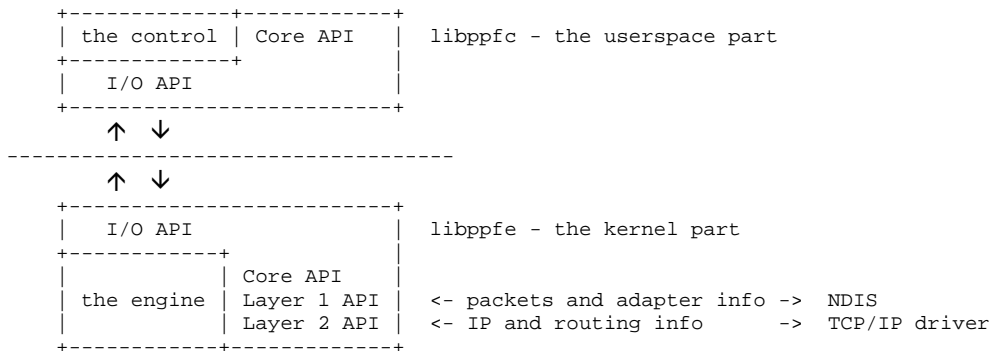
The library consists of two principal components - the kernel module and the userspace module.

The kernel module implements the features listed in *previous* section, which comprise an *engine* environment. The userspace module implements a *control* environment. As their names suggest, the engine code implements an actual packet processing logic, and the control code configures and controls the engine.

The kernel part of *libppf* is further referred to as *libppfe*, and the userspace part - *libppfc*.

Both *libppfe* and *libppfc* implement IO API, which enables the engine and the control code to communicate with each other.

The following diagram depicts high-level structure of *libppf*.



Core API creates a cross-platform foundation for all other *libppf* APIs and its code. This API hides compiler- and platform-specific details pertaining to structure packaging directives, commonly used integral types, byte ordering, memory management and debugging support.

Layer 1 API wraps the code responsible for interfacing the network stack, intercepting inbound and outbound traffic and monitoring a presence and the status of the network adapters.

Layer 2 API defines methods for tracking per-adapter IP information and IP routing.

## Libppfe

Libppfe is implemented in a form of a kernel driver, which can be loaded either during the boot sequence or at explicit application request. Packets are intercepted using so-called NDIS hooking technique, which enables almost instantaneous driver installation and removal, especially when compared to the filters in a form of Intermediate NDIS drivers.

The driver code comes with two unresolved external references. These two methods are the points of entry into the engine code; first is used during an initialization sequence, and second - during the shutdown.

Upon loading, the driver performs required *libppfe* initialization and passes the execution to the engine code. The engine code initializes control IO channel and/or initiates a packet interception and then returns.

Once initialized the engine's execution becomes purely event-driven. *Libppfe* delivers events and notifications via a set of callbacks, which the engine code registers with the library. These events and notifications include:

- intercepted network packets
- network interfaces going up and down
- IP addresses being assigned and unassigned and
- routing table being modified
- IO packets received from the control code

The shutdown process is initiated by the driver's DriverUnload entry, which first passes the control to the engine shutdown handler and then cleans up and shutdowns *libppfe* itself.

## Libppfc

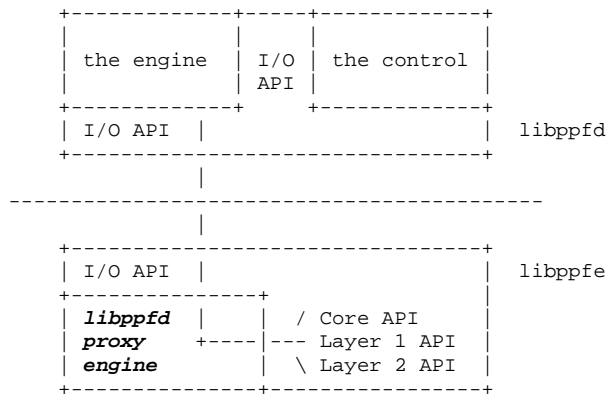
*Libppfc* is a userspace library, which defines and implements IO API. The main application first initializes *libppfc* with a set of callbacks, which are then used to deliver the data sent from the engine and to indicate the status of the IO channel.

Internally *libppfc* opens a file object exposed by *libppfe* driver and uses Win32 API to read/write the data from/to it.

## Libppfd

*Libppfd* (debug) is a special debugging version of the library. This version emulates the engine environment in the userspace and allows developing, compiling and debugging the engine code with conventional (non-kernel) development tools.

The following diagram shows *libppfd* structure and how it relates to *libppf*:



The kernel part of *libppfd* runs in an engine environment and simply relays event and API calls between *libppfe* and the userspace. It essentially implements a simple bi-directional RPC protocol over *libppf*'s IO channel. At the other end of the channel there is a *libppfd*'s userspace module, which creates an exact replica of an engine environment in the userspace. The module also provides both environments with respective implementations of IO API, enabling the engine and the control code to communicate as if they were using original *libppf*.

*Libppfd*'s kernel code does not change. Therefore it can be built once and then used in a binary form to completely eliminate the need for any kernel development whatsoever.

However, the overhead of RPC protocol makes *libppfd*-based applications unsuitable for a time-sensitive or a high-volume packet processing tasks, which is why *libppfd* is positioned as a *debugging* version of *libppf*.

Fortunately, since *libppfd* emulates exact engine environment, the engine code can be *seamlessly* moved to the kernel once it's fully debugged and is ready for the prime-time.

## Information

The library is available for a commercial licensing, *libppfd* version is available for a free non-commercial use. For more information and downloads please visit <http://libppf.cipherica.com>

## Appendix A.1 - libppfe API

```
/*
 * Layer 2 (datalink) API callbacks
 */
typedef struct x4_l2_device
{
    char * name;                /* system name */
    char * desc;               /* user-friendly name if available */

    x4s_net_media media;       /* media information */

    uint mtu;                  /* maximum transmission unit (bytes) */
    uint bps;                  /* link speed in bits per second */
} x4s_l1_device;

/* */
typedef struct x4_l2_config
{
    void * (*up) (const x4s_l2_device * );
    void (*down) (void *);

    void (*send) (void *, x4s_packet * ); /* intercepted outbound packet */
    void (*recv) (void *, x4s_packet * ); /* intercepted inbound packet */
} x4s_l2_config;

/*
 * Layer 3 (IP) API callbacks
 */
typedef struct x4_l3_config
{
    void (*assign) (void *, const x4s_net_ip * ip);
    void (*unassign)(void *, const x4s_net_ip * ip);

    void (*purge) ();          /* callee must purge its cache of routes */
} x4s_l3_config;

/*
 * IO API callbacks
 */
typedef struct x4_io_config
{
    void (*open) ();
    void (*close) ();
    void (*recv) (const void * data, uint len);

    void (*can_send)();
} x4s_io_config;

/*
 * Layer 2 API
 */
void x4_l2_init(const x4s_l2_config *);
void x4_l2_term();

void x4_l2_enum();

bval x4_l2_send(const x4s_l2_device *, x4s_packet *);
bval x4_l2_recv(const x4s_l2_device *, x4s_packet *);

/*
 * Layer 3 API
 */
void x4_l3_init(const x4s_l3_config *);
void x4_l3_term();

void * x4_l3_route(const x4s_net_ip * peer, x4s_net_ip * next_hop);
```

```

/*
 * IO API
 */
bval x4_io_init(const x4s_io_config * );
void x4_io_term();

bval x4_io_send(const void * data, uint len);
void x4_io_error(int);

/*
 * Main entry points
 */
bval x4_ppfe_init();
void x4_ppfe_term();

```

## Appendix A.2 - libppfc API

```

/*
 * IO callbacks
 */
typedef struct x4_ppfc_config
{
    void (*io_error)(int);
    void (*io_recv)(const void * data, uint len);

    void (*io_can_send)();
} x4s_ppfc_config;

/*
 * IO API
 */
bval x4_ppfc_init(const x4s_ppfc_config * );
void x4_ppfc_term();

bval x4_ppfc_send(const void * data, uint len);

```