README.concap.txt

Description of the "concap" encapsulation protocol interface

The "concap" interface is intended to be used by network device drivers that need to process an encapsulation protocol.
It is assumed that the protocol interacts with a linux network device by
- data transmission
- connection control (establish, release)

Thus, the mnemonic: "CONnection CONtrolling eNCAPsulation Protocol".

This is currently only used inside the isdn subsystem. But it might also be useful to other kinds of network devices. Thus, if you want to suggest changes that improve usability or performance of the interface, please let me know. I'm willing to include them in future releases (even if I needed to adapt the current isdn code to the changed interface).

Why is this useful?

The encapsulation protocol used on top of WAN connections or permanent point-to-point links are frequently chosen upon bilateral agreement. Thus, a device driver for a certain type of hardware must support several different encapsulation protocols at once.

The isdn device driver did already support several different encapsulation protocols. The encapsulation protocol is configured by a user space utility (isdnctrl). The isdn network interface code then uses several case statements which select appropriate actions depending on the currently configured encapsulation protocol.

In contrast, LAN network interfaces always used a single encapsulation protocol which is unique to the hardware type of the interface. The LAN encapsulation is usually done by just sticking a header on the data. Thus, traditional linux network device drivers used to process the encapsulation protocol directly (usually by just providing a hard_header() method in the device structure) using some hardware type specific support functions. This is simple, direct and efficient. But it doesn't fit all the requirements for complex WAN encapsulations.

> The configurability of the encapsulation protocol to be used makes isdn network interfaces more flexible, but also much more complex than traditional lan network interfaces.

Many Encapsulation protocols used on top of WAN connections will not just stick a header on the data. They also might need to set up or release the WAN connection. They also might want to send other data for their private purpose over the wire, e.g. ppp does a lot of link level negotiation before the first piece of user data can be transmitted. Such encapsulation protocols for WAN devices are typically more complex than encapsulation protocols for lan devices. Thus, network interface code for typical WAN devices also tends to be more complex.

In order to support Linux’ x25 PLP implementation on top of isdn network interfaces I could have introduced yet another branch to the various case statements inside drivers/isdn/isdn_net.c. This eventually made isdn_net.c even more complex. In addition, it made isdn_net.c harder to maintain. Thus, by identifying an abstract interface between the network interface code and the encapsulation protocol, complexity could be reduced and maintainability could be increased.

Likewise, a similar encapsulation protocol will frequently be needed by several different interfaces of even different hardware type, e.g. the synchronous ppp implementation used by the isdn driver and the asynchronous ppp implementation used by the ppp driver have a lot of similar code in them. By cleanly separating the encapsulation protocol from the hardware specific interface stuff such code could be shared better in future.

When operating over dial-up-connections (e.g. telephone lines via modem, non-permanent virtual circuits of wide area networks, ISDN) many encapsulation protocols will need to control the connection. Therefore, some basic connection control primitives are supported. The type and semantics of the connection (i.e the ISO layer where connection service is provided) is outside our scope and might be different depending on the encapsulation protocol used, e.g. for a ppp module using our service on top of a modem connection a connect_request will result in dialing a (somewhere else configured) remote phone number. For an X25-interface module (LAPB semantics, as defined in Documentation/networking/x25-iface.txt) a connect_request will ask for establishing a reliable lapb datalink connection.

The encapsulation protocol currently provides the following service primitives to the network device.

- create a new encapsulation protocol instance
- delete encapsulation protocol instance and free all its resources
- initialize (open) the encapsulation protocol instance for use.
- deactivate (close) an encapsulation protocol instance.
- process (xmit) data handed down by upper protocol layer
- receive data from lower (hardware) layer
- process connect indication from lower (hardware) layer
- process disconnect indication from lower (hardware) layer

The network interface driver accesses those primitives via callbacks provided by the encapsulation protocol instance within a struct concap_proto_ops.

```
struct concap_proto_ops{

        /* create a new encapsulation protocol instance of same type */
        struct concap_proto *  (*proto_new) (void);
        /* delete encapsulation protocol instance and free all its resources.
           cprot may no longer be referenced after calling this */
        void (*proto_del)(struct concap_proto *cprot);

        /* initialize the protocol's data. To be called at interface startup
           or when the device driver resets the interface. All services of the
           encapsulation protocol may be used after this*/
        int (*restart)(struct concap_proto *cprot,
                       struct net_device *ndev,
                       struct concap_device_ops *dops);

        /* deactivate an encapsulation protocol instance. The encapsulation
           protocol may not call any *dops methods after this. */
        int (*close)(struct concap_proto *cprot);

        /* process a frame handed down to us by upper layer */
        int (*encap_and_xmit)(struct concap_proto *cprot, struct sk_buff *skb);

        /* to be called for each data entity received from lower layer*/
        int (*data_ind)(struct concap_proto *cprot, struct sk_buff *skb);

        /* to be called when a connection was set up/down.
           Protocols that don't process these primitives might fill in
           dummy methods here */
        int (*connect_ind)(struct concap_proto *cprot);
        int (*disconn_ind)(struct concap_proto *cprot);
};
```

The data structures are defined in the header file include/linux/concap.h.

A Network interface using encapsulation protocols must also provide
some service primitives to the encapsulation protocol:

- request data being submitted by lower layer (device hardware)
- request a connection being set up by lower layer
- request a connection being released by lower layer

The encapsulation protocol accesses those primitives via callbacks
provided by the network interface within a struct concap_device_ops.

```
struct concap_device_ops{

        /* to request data be submitted by device */
        int (*data_req)(struct concap_proto *, struct sk_buff *);

        /* Control methods must be set to NULL by devices which do not
           support connection control. */
        /* to request a connection be set up */
        int (*connect_req)(struct concap_proto *);

        /* to request a connection be released */
        int (*disconn_req)(struct concap_proto *);
};
```

The network interface does not explicitly provide a receive service
because the encapsulation protocol directly calls netif_rx().

An encapsulation protocol itself is actually the

```
struct concap_proto{
        struct net_device *net_dev;             /* net device using our service
*/
        struct concap_device_ops *dops; /* callbacks provided by device */
        struct concap_proto_ops  *pops; /* callbacks provided by us */
        int flags;
        void *proto_data;               /* protocol specific private data, to
                                           be accessed via *pops methods only*/
        /*
          :
          whatever
          :
          */
};
```

Most of this is filled in when the device requests the protocol to
be reset (opend). The network interface must provide the net_dev and
dops pointers. Other concap_proto members should be considered private
data that are only accessed by the pops callback functions. Likewise,
a concap proto should access the network device's private data
only by means of the callbacks referred to by the dops pointer.

A possible extended device structure which uses the connection controlling
encapsulation services could look like this:

```
struct concap_device{
        struct net_device net_dev;
        struct my_priv  /* device->local stuff */
                        /* the my_priv struct might contain a
                           struct concap_device_ops *dops;
                           to provide the device specific callbacks
                        */
        struct concap_proto *cprot;        /* callbacks provided by protocol */
};
```

Misc Thoughts
=============

The concept of the concap proto might help to reuse protocol code and
reduce the complexity of certain network interface implementations.
The trade off is that it introduces yet another procedure call layer
when processing the protocol. This has of course some impact on
performance. However, typically the concap interface will be used by
devices attached to slow lines (like telephone, isdn, leased synchronous
lines). For such slow lines, the overhead is probably negligible.
This might no longer hold for certain high speed WAN links (like

ATM).

If general linux network interfaces explicitly supported concap protocols (e.g. by a member struct concap_proto* in struct net_device) then the interface of the service function could be changed by passing a pointer of type (struct net_device*) instead of type (struct concap_proto*). Doing so would make many of the service functions compatible to network device support functions.

e.g. instead of the concap protocol's service function

```
  int (*encap_and_xmit)(struct concap_proto *cprot, struct sk_buff *skb);
```

we could have

```
  int (*encap_and_xmit)(struct net_device *ndev, struct sk_buff *skb);
```

As this is compatible to the dev->hard_start_xmit() method, the device driver could directly register the concap protocol's encap_and_xmit() function as its hard_start_xmit() method. This would eliminate one procedure call layer.

The device's data request function could also be defined as

```
  int (*data_req)(struct net_device *ndev, struct sk_buff *skb);
```

This might even allow for some protocol stacking. And the network interface might even register the same data_req() function directly as its hard_start_xmit() method when a zero layer encapsulation protocol is configured. Thus, eliminating the performance penalty of the concap interface when a trivial concap protocol is used. Nevertheless, the device remains able to support encapsulation protocol configuration.