STREAMS

Most of the device driver code is similar. Device independent part do buffering and basic data processing. Char devices has message boundaries.

Network implementation consists of many layers having similar task, packet rewriting and sending to another layer. All buffer management, message boundary management, queuing code is similar.

For code re-usability a modular design of devices is required. STREAMS subsystem solves this problem. A stream device can be constructed by combination of modules.

- message based interface
- buffer management
- flow control
- priority management

Stream head is the part in the user application side. Each module contains a pair of queues, read and write queues, upstream and downstream respectively.

Streams also supports multiplexing:
TCP, UDP upper multiplexor.
IP two way multiplexor.
Lower multiplexor: multiple modules at the bottom.

Modules choose flow control of messages.

Message structure:

Message 1

Message 2
struct msb or mblk_t type contains the message information
struct datab or dblk_t type contains the buffer information for the messages.

b_rptr and b_wptr pointers keep the actual part used by the message from the buffer. (Since buffer is a dynamic entity, it may grow as written and shrinks as read). Also two messages sharing same buffer is possible for storage economy. (dblk_t.db_ref counts the references).
This is called virtual copying. For example TCP layer keeps a copy of the message as it is sending it through the downstream for retransmission.

Modules:

Each module has a pair of queues, read and write queues. Having:

q_qinfo : points to qinit structure containing procedural interface of the queue
q_first,q_last: double linked list of messages in the queue
q_next: next queue (upstream or downstream)
q_hiwat, q_lowat: watermark of messages..

Each module queue points to a struct qinit containing service data for the queue:
qi_putp: put() function pointer
qi_srvp: service() function pointer
qi_qopen: open() function pointer
qi_qclose: close() function pointer
qi_minfo: module information
qi_mstat: module statistics

Module interaction is in terms of transferring messages to the next queue.
putnext() call puts the message to the next queue, calls put() function of the next queue.

Each module implements service() procedure which is executed asynchronously to handle messages.

runqueues() function is called by kernel whenever a stream operation is done. Service routines are activated according to queue statuses.

Flow control is achieved by the service functions. and low-high water marks of the current and target queues.

Pushing modules:

Modules can be pushed onto an open stream with ioctl with I_PUSH command. Also autopush mechanism exists for automatically pushing modules to a stream driver at first open.
ioctl(fd, I_PUSH,"module name");
Multiplexing:

Upper multiplexors (i.e. minor devices for a device).
Lower multiplexors (i.e. pseudo device communicating with multiple devices)

creating multiplexors.

```c
fd_enet=open("/dev/enet",O_RDWR);
fd_fddi=open("/dev/fddi",O_RDWR);
fd_ip=open("/dev/ip",O_RDWR);
ioctl(fd_ip, I_LINK, fd_enet);
ioctl(fd_ip, I_LINK, fd_fddi);
```