

## Acknowledgement

I feel pleasure to express my heartfelt gratitude to my Guide Dr.P.C.Saxena for his uncompromising guidance constant supervision and constructive criticism without which this work would not have been completed successfully.

I extend my sincere thanks to Prof.R.G.Gupta Dean School of Computer & System Sciences, Jawahar Lal Nehru University For his encouragement and facilities for the completion of this work.

I also take this opportunity to thank all faculty and staff members and my friends who have been directly or indirectly helpful in eliminating a variety of problems encountered by me in the course of completion of this dissertation.


Devesh Gupta  
(Devesh Gupta)


## Certificate

This is to certify that the Dissertation entitled "Character Device Driver for Unix", being submitted by me to Jawahar Lal Nehru University in the partial fulfillment of the requirement for the award of degree of Master of Technology, in Computer Science is a record of original work done by me under the supervision of Dr. P.C.Saxena, Associate Professor, School of Computers and Systems Sciences, Jawahar Lal Nehru University during the Year 1992 Monsoon Semester.

The results reported in this dissertation have not been submitted in part or full to any other university or Institute for the award of any degree or diploma, etc.

  
(Devesh Gupta)

  
(Prof. R.G. Gupta)  
Dean, <sup>31/12/92</sup>  
School of Computer &  
Systems Sciences.  
JNU, New Delhi.

  
(Dr. P.C. Saxena)  
Associate Professor,  
School of Computer &  
Systems Sciences.  
JNU, New Delhi.

## CONTENTS

	<b>ACKNOWLEDGEMENTS</b>	<b>PAGE NO.</b>
	<b>PREFACE</b>	
<b>CHAPTER-I</b>	INTRODUCTION	1-2
	INTRODUCTION TO PROBLEM DESCRIPTION	1 1
<b>CHAPTER-II</b>	OVER-VIEW OF STREAMS	3-16
	WHAT IS STREAMS	3
	STREAMS COMPONENTS	3
	MESSAGES	6
	MESSAGES TYPE	6
	MESSAGE QUEUING PRIORITY	8
	MODULES	9
	STREAM CONSTRUCTION	9
	OPENING A STREAM DEVICE FILE	13
<b>CHAPTER-III</b>	ALGORITHM FOR SIMPLER MODULES & MODULES INSERTION	17-24
	ADDING AND REMOVING MODULES	17
	CLOSING THE STREAM	18
	INSERTING MODULES	19
	MODULE AND DRIVER CONTROL	21

<b>CHAPTER-IV</b>	DESCRIPTION & ALGORITHM FOR SIMPLE DRIVERS	25-37
	LOOP AROUND DRIVER	25
	ALGORITHM LOOP AROUND DRIVER	26
<b>CHAPTER-V</b>		38-48
	CONCLUSIONS & FINDINGS	38
	DRIVER CONFIGURATION	39
	WRITING A DRIVER	40
	RULES OF DRIVER DEVELOPMENT	41
	MAJOR & MINOR DEVICE NUMBERS	43
	STREAMS DRIVERS	46
<b>APPENDIX-I</b>	LISTING OF PROGRAM	49-72
<b>BIBLIOGRAPHY</b>		1-3

## PREFACE

In the present work I have tried to explain how to write device drivers using streams facility of unix., and implementation for character device driver for unix. Chapter I **Introduction** deals with need for Device Drivers. Chapter II **"Overview of Streams"** tells about structure of stream facilities provided in "stream" and about the components of streams.

Chapter III **Algorithm for "simpler Modules & modules Insertion"**. deals with Adding and Removing Modules. How to close the stream Plus figure and algorithm for case converter module (this module changes lower case letters to upper case i.e. a --> A b -->B etc). Chapter IV deals with **"Algorithm for simple Drivers"**. Drivers described here is the loop Around Driver. Loop around driver loops data from one open stream to another open stream. Chapter describes how to open another stream Chapter describes how two streams are opened and how message is transferred from one stream to other. Chapter also gives algorithm for loop around drivers with simultaneous description.

Chapter V **Conclusions & Findings.** Describes the steps how to write device driver, and then the rules for driver development, modification to be made in file system, major & minor device no, etc.

**Appendix - 1** deals with modules required in character device driver implementations. Bibliography at the end has been added for additional reference of the reader of this dissertation.

## CHAPTER I

### INTRODUCTION

The Fundamental work in adapting UNIX to new hardware environment consists largely in providing, appropriate device drivers, If the new h/w is similar to the old, then existing driver can readily be adapted. The driver developer must take into account manner in which data is communicated with the h/w device and manner in which asynchronous signals such as interrupts are communicated. Since drivers are implementing system calls and often must respond to low-level interrupts, they must interface intimately with the UNIX kernel. In the present work I'll be writing character device driver using streams facility of Unix.

The streams I/O system provides a background for developing drivers allowing them to be structured in simple and understandable (well, almost) modules and allowing these modules to be modified dynamically while the system is operational. In particular it unifies the interface between the kernel and the user specifying rather exactly the forms of dialogue that can occur.

#### **Description:-**

Streams was created by Dennis Ritchie, the co-creator of UNIX, and first described in an article "A stream Input-Output System" in the October 1984 AT & T Bell laboratories technical Journal. It became an integral part of system V

UNIX Release 3.0 which was first made publicly available in June 1986.

Ritchie described the STREAMS interface as "flexible-coroutine-based design which replaces the traditional rigid connection between processes and terminals or networks". (and describes it as running on "about 20 machines in the Information Science Research Division of A T & T Bell laboratories"). The currently distributed version of streams provide essentially the interface described in the paper but adds a provision for multiplexing several modules into one. Ritchie discusses this problems and states

".....a general multiplexing mechanism could help..., but again I do not yet know how to design it".

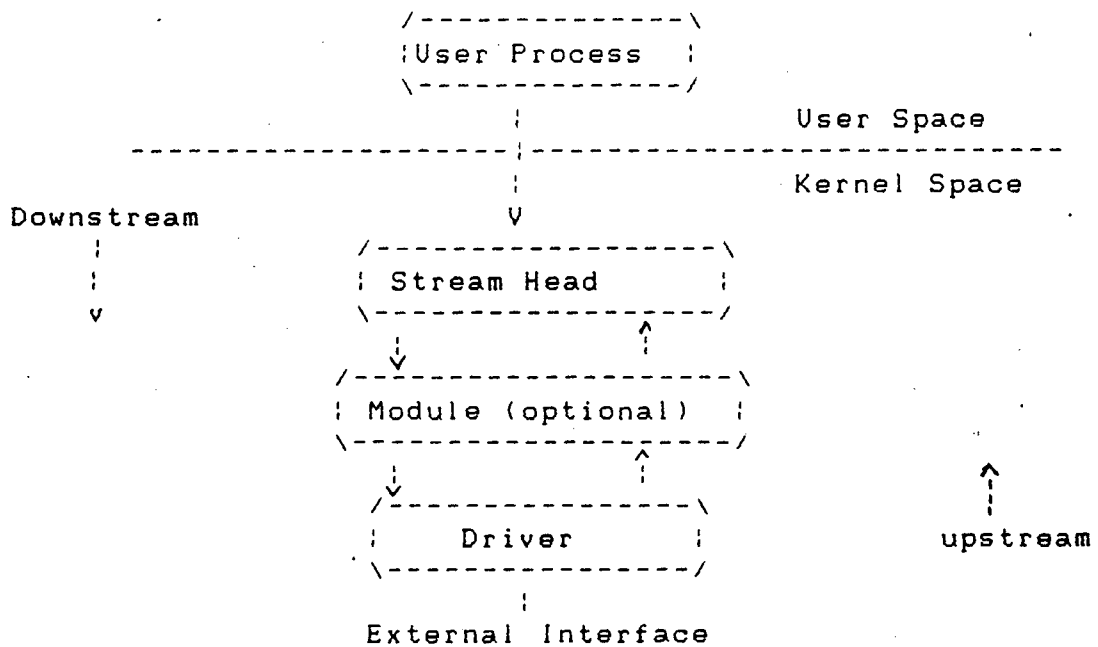
A streams is a collection of units called modules", Providing at one end the head, a user kernel interface following a prescribed protocol, and at the other end, the driver, providing what is usually very similar to a traditional device driver. The modules between the head and the driver serve as filters, transforming data as required, as servers, implementing particular data communication protocols; or as routers, choosing lower streams for routing data in and out of various devices and choosing upper streams for routing data to and from various processes. Any module may send and receive message of its own, representing control requests or response or error indications. Following work is based on developing algorithm for device drivers.





opened or the module is pushed (added) on to the stream.

Figure : Simple Stream



Data are passed between a driver and the stream head and between modules in the form of messages. A message is a set of data structure used to pass data, status, and control information between user process, modules and drivers. Messages that are passed from the device are said to travel downstream (also called write side). Similarly, message passed in the other direction, from the device to the process or from the driver to the stream head, travel upstream (also called read-side).

A streams message is made up of one or more message blocks. Each block is a 3-tuple consisting of a header, a data block and a data buffer. The stream head

transfers data between the data space of a user process and STREAMS Kernel data-space. Data to be sent to a driver from a user process are packaged into STREAMS message and passed downstream when a message consisting data arrives at the stream head from down stream, the message is processed by the stream head, which copies the data into user buffers.

### STREAMS COMPONENTS 1-

#### Queue1-

A queue is an interface between a STREAMS driver or module and the rest of the stream. Queues are always allocated as an adjacent pair. The queue, and the lower address in the pair is a read queue, and the queue with the higher address is used for the write queue.

A queue's service routine is invoked to process messages on the queue. Each queue also has a pointer to an open and close routine. The open routine of a driver is called when the driver is first opened and on every successive open of the stream. The close routine module is called when the module is first pushed on the stream and on every successive open of the stream. The close routine of the driver is called when the last reference to the stream is dismantled.

## MESSAGES | -

All input and output under STREAMS is based on messages. The object passed between STREAMS modules are pointers to messages. All STREAMS messages use two data structures (msgb and datab) to refer to the message data. These data structures describes the type of the message and contain pointers to the data of the message, as well as other information. Messages are sent through a stream by successive calls to the put procedure of each module or driver in the stream.

## MESSAGE TYPE | -

All STREAMS messages assigned message types to indicate their intended use by modules and drivers and to determine their handling by the Stream head. A driver or module can assign most types to a message it generates, and a module can modify a message type during processing. The Streams head will convert certain system calls to specified message types and send them downstream, and it will respond to other calls by copying the contents of certain message types that were sent upstream.

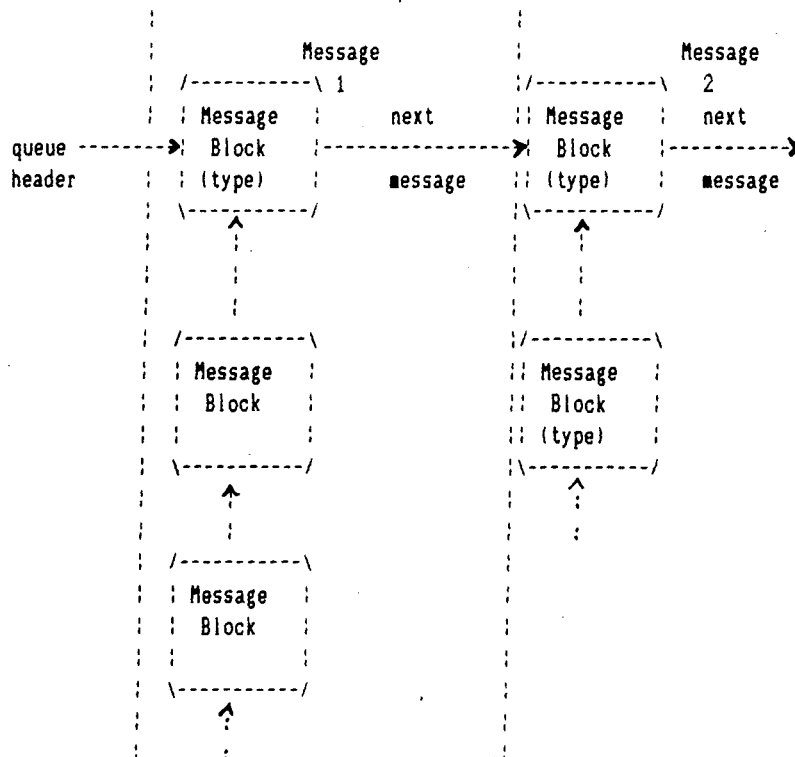
Most message types are internal to STREAMS and can only be passed from one STREAMS component to another. A few message types, for example M\_DATA, M\_PROTO, and M\_PCROPOTO, can also be passed between a Stream and user processes. M\_DATA message carry data within a Stream and

between a Stream and a user process. M\_PROTO or M\_PCROTO message carry both data and control information.

As shown in the figure, a STREAMS message consists of one or more linked message blocks that are attached to the first message block of the same message.

Messages can exist stand-alone, as in the figure, when the message is being processed by a procedure. Alternately, a message can await processing on a linked list of messages, called a message queue. In next figure Message 2 is linked to message one.

Figure : Messages on a Message Queue



When a message is on a queue, first block of the message contains links to preceding and succeeding messages on the same message queue, in addition to the link to the second block of the message (if present). The message queue head and tail are contained in the queue.

STREAMS utility routines enable developers to manipulate queue.

### Message Queuing Priority

In certain cases, messages containing urgent information (such as a break or alarm conditions) must pass through the Stream quickly. To accommodate these cases, STREAMS provides multiple classes of message queuing priority. All messages have an associated priority field. Normal (ordinary) messages have a priority of zero. Priority messages have a priority greater than zero. High priority messages are high priority by virtue of their message type. The priority field in high priority messages is unused and should always be set to zero.

Non priority, ordinary messages are placed at the end of the queue following all other messages in the queue. Priority messages can be either high priority or priority band messages. High priority messages are placed at the head of the queue but after any other high priority messages already in the queue. Priority band messages

that enable support of urgent, expedited data are placed in the queue after high priority messages but before ordinary messages.

Message priority is defined by the message type; once a message is created, its priority cannot be changed. Certain message types come in equivalent high priority/ordinary pairs (for example M\_PCPROTO and M\_PROTO), so that a module or device driver can choose between the two priorities when sending information.

### MODULES

A module performs intermediate transformations on messages passing between a Stream head and a driver. There may be zero or more modules in a stream (zero when the driver performs all the required character and device processing).

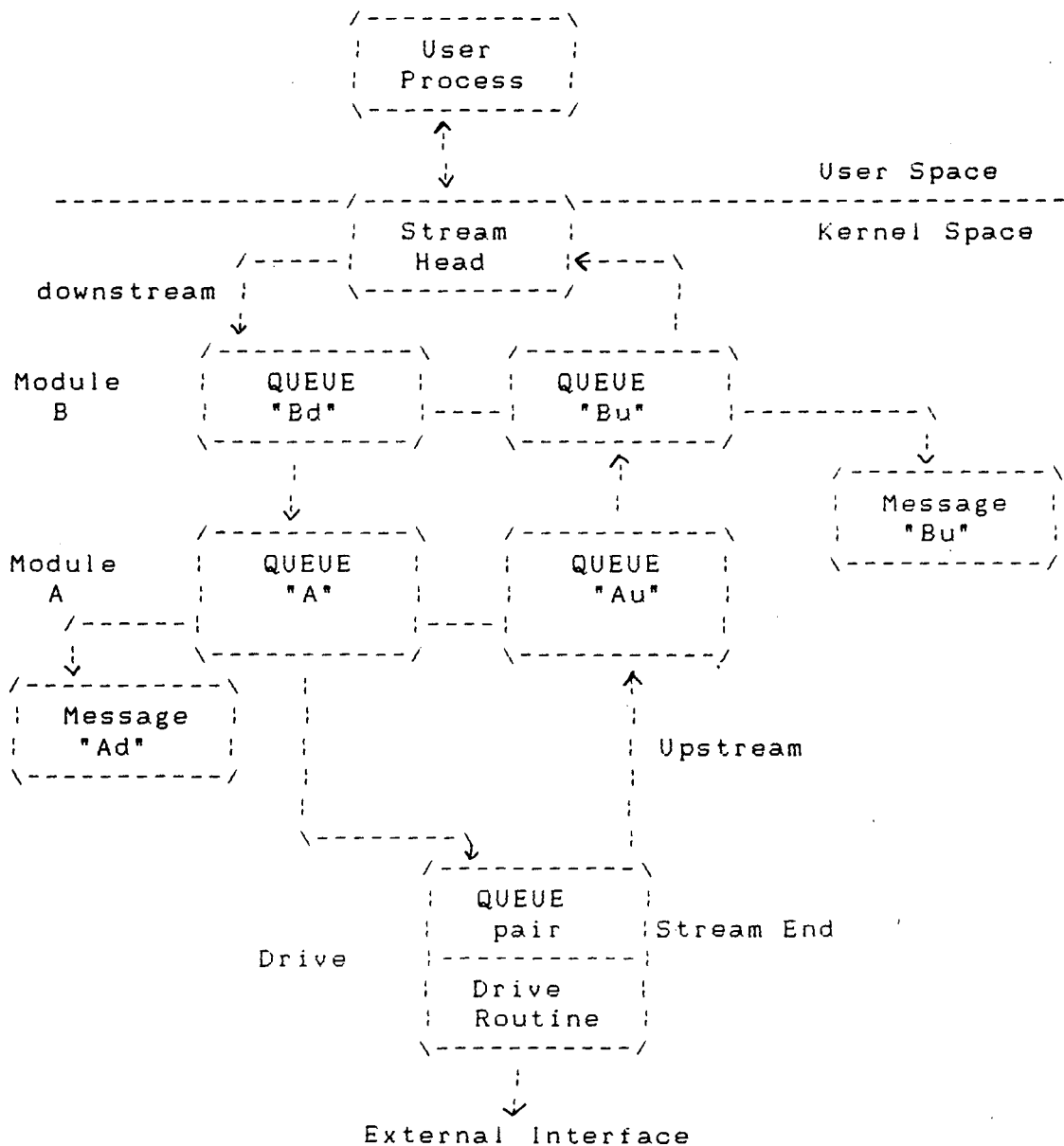
each module is constructed from a pair of queue structures. One queue performs functions on messages passing upstream through the module. The other set performs another set of functions on downstream messages. Each queue can directly access the adjacent queue in the direction of message flow. In addition, within a module, a queue can readily locate its mate and access its message and data.

### **STREAM CONSTRUCTION**

STREAMS constructs a Streams as a linked list of kernel resident data structures. The list is created as a

set of linked queue pairs. The first queue pair is the head of the Stream and the second queue pair is the end of the Stream. The end of the Stream presents a device driver, pseudo device, driver, or the other end of a STREAMS-based pipe. Kernel routines interface with the Stream head to

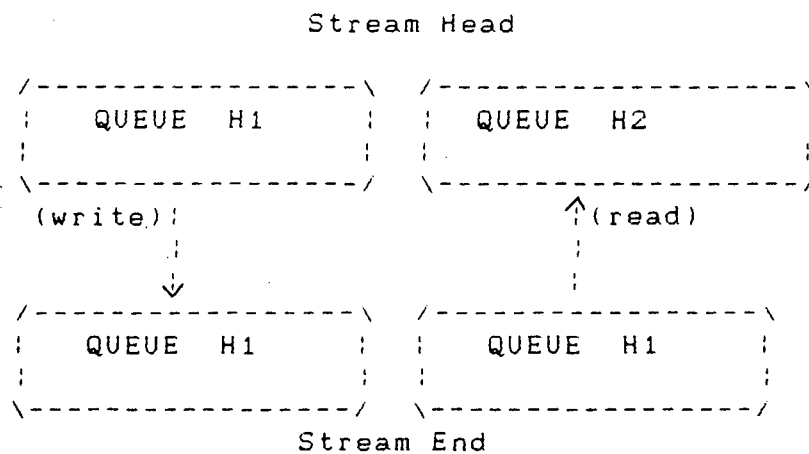
Figure: A Stream in More Detail





perform operations on the Stream. Figure depicts the upstream (read) and downstream (write) portions of the stream. Queue H2 is the upstream half of the Stream head and queue H1 is the downstream half of the Stream head. Queue E2 is the upstream half of the Stream end and queue E1 is the downstream half of the Stream end.

Figure : Upstream and Downstream Stream Construction



of the entry pint, a procedure to process any message received by the queue. The procedures for queues H1 and H2, process messages received by the other end of the Stream the Stream end )tail). Messages move from one end to the other, from one queue to the next linked queue, as the procedure specified by that queue is executed.

Figure shows the data structure forming each queue: queue, qinits, qband, module\_info, and module\_stat. The qband structure have information for each priority band in the queue. The queue data structure contains various modifiable values for the queue. The qinit structure contains a pointer to the processing procedures, the module\_

'info structure contains initial limit values, and the module\_stat structure is used for statistics gathering. Each queue in the queue pair contains a different set of these data structures. There is a queue, qinit, module\_info, and module\_stat data structure for the upstream portion of the queue pair and a set of data structures of the downstream portion of the pair. In some situations, a queue pair may share some or all of the data structures. For example, there may separate qinit structure for each queue in the pair and one module\_stat structure that represents both queues in the pair. Figure shows two neighboring queue pairs with links (solid vertical arrows) in both directions. When a module pushed is pushed into a Stream, STREAMS creates a queue pair and links each queue in the pair to its neighboring queue in the upstream and downstream direction. The linkages allows each queue to locate its next neighbor. This relations is implemented between adjacent queue pairs by the q\_next pointer. Within a queue pair, each queue locates its mate by use of STREAMS macros, since there is no pointer between

Figure : Stream Queue Relationship

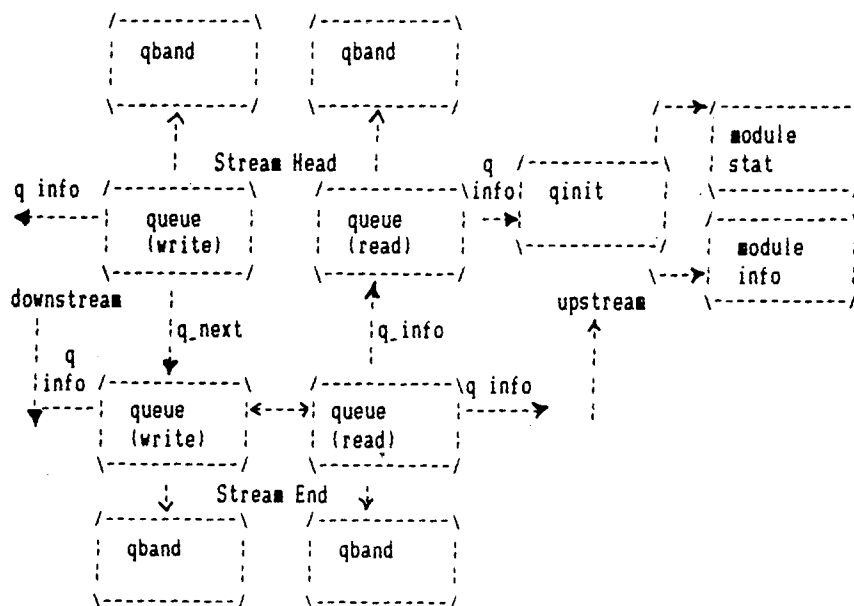


figure shows two neighboring queue pairs with links (solid vertical arrows) in both directions. when a module is pushed into a Stream, STREAMS creates a queue pair and links each queue in the pair to its neighboring queue in the implemented between adjacent queue pairs by the q\_next pointer. Within a queue pair, each queue locates its mate by use of STREAMS macros, since there is no pointer between the two queue procedures only as destinations towards which messages are sent.

### **Opening a STREAMS Device File**

One way to construct a Stream is to open a STREAM-based driver file. All entry points into the driver are defined by the streamtab structure for that driver. The streamtab structure has a format as follows.

#### **Struct streamtab**

```
Struct qinit      *st_rdinit
Struct qinit      *st_wrinit;
Struct qinit      *st_mazrinit;
Struct qinit      *st_muxwinit;
```

The streamtab structure defines a module or driver. st\_rdinit points to the read qinit structure for the driver and st\_wdinit points to the driver's write qinit structure. st\_muxwinit point to the lower read and write qinit structures if the driver is a multiplexer driver.

If the open call is the initial file open, a Stream is created. (There is one Stream per major/minor device pair).

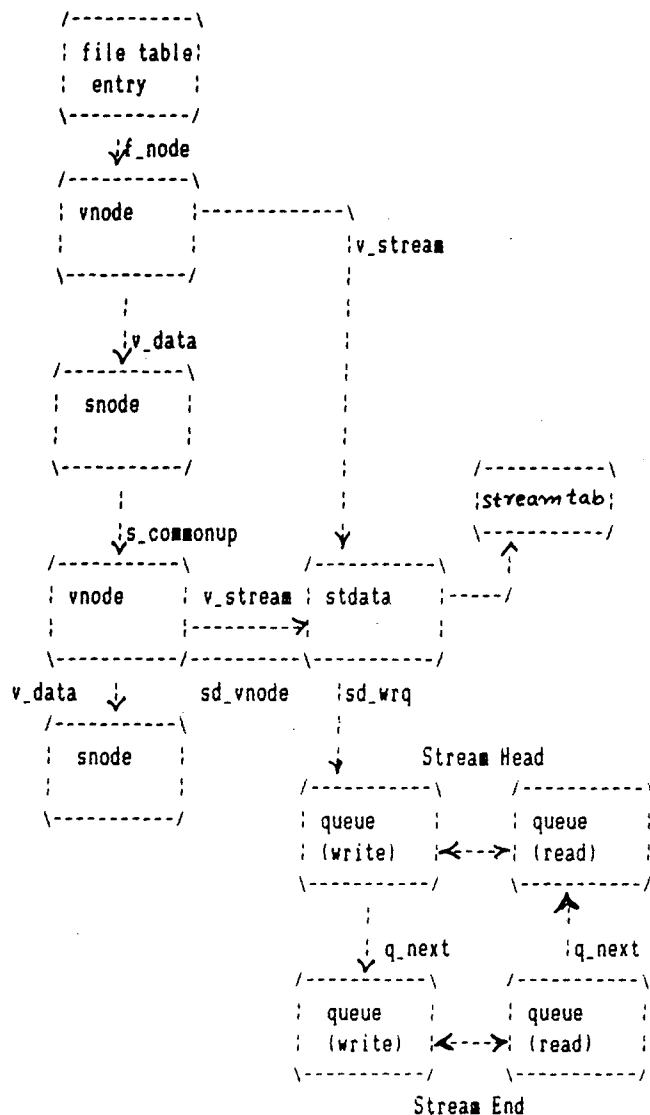
First, an entry is allocated in the user's file table and a vnode is created to represent the opened file. The file table entry is initialized to point to the allocated vnode and the vnode is initialized to specify a file of type character special.

Second, a Stream header is created from an stdata data structure and a Stream head is created from a pair of queue structures. The content of stdata and queue are initialized with predetermined values, including the Stream head processing procedures.

The snode contains the file system dependent information. It is associated with the vnode representing the device. The s\_commonvp field of the snode points to the common device vnode. The vnode field, v\_data, contains a pointer to the snode. Instead of maintaining a pointer to the vnode, the snode contains the vnode as an element. The sd\_vnode field of stdata is initialized to point to the allocated vnode. The v\_stream field of the vnode data structure is initialized to point to the Stream header, thus there is a forward and backward pointer between the Stream header and the vnode. There is one Stream header per Stream. The header is used by STREAMS while performing operations on the Stream. The header is used by STREAMS while performing operations on the Stream. In the downstream portion of the Stream, the Stream header points to the downstream half of the Stream head queue pair. Similarly, the upstream portion of the

of the Stream terminates at the Stream header, since the upstream half of the Stream head queue pair points to the onward, a Stream is constructed of linked, queue pairs. Next, a queue structure pair is allocated for the driver. The queue limits are initialized to those values specified in the corresponding module info structure. The queue processing routines are initialized to those specified by the corresponding qinit structure. Finally, the driver open procedure (located via its read qinit structure) is called.

Figure: Opened STREAMS-based Driver



modules on the Stream. When a Stream is already open, routines of all modules and the driver on the Stream being called. Note that this is an reverse order from the way a Stream is initially set up. That is, a driver is opened and a module is pushed on a Stream. When a push occurs the module open routine is called. If another open of the same device is made, the open, routine of the module will be called followed by the open routine of the driver. This is opposite from the initial order of opens when the Stream is created.

## Chapter-3

### Algorithm for Simpler Modules & Modules Insertion

#### **Adding and Removing Modules**

As part of constructing a Stream a module can be added (pushed) with an ioctl I\_PUSH system call. The push inserts a module beneath the Stream head. Because of the similarity of STREAMS components, the push operation is similar to the driver open. First, the address of the qinit structure for the module is obtained.

Next, STREAMS allocates a pair of queue structures and initializes their contents as in the driver open.

Then, q\_next values are set and modified so that the module is interposed between the Stream head and its neighbor immediately downstream. Finally, the module open procedure (located via qinit) is called.

Each push of module is independent, even in the same Stream. If the same module is pushed more than once on a Stream there will be multiple occurrences of that module in the Stream. The total number of pushable modules that may be contained on any one Stream is limited by the Kernel parameter NSTRPUSH .

An ioctl I\_POP system call revokes (pops) the module immediately below the Stream head. The pop calls the module close procedure. On return from the module close, any messages left on the module's message queues are freed (deallocated). Then, STREAMS connects the Stream head to the

component previously below the popped module and deallocates the module's queue pair. I\_PUSH and I\_POP enable a user process to dynamically alter the configuration of a Stream by pushing and popping modules as required. For example, a module maybe removed and a new one inserted below the Stream head. Then the original module can be pushed back after the new module has been pushed.

### **CLOSING THE STREAM**

The last close to a STREAMS file dismantles the Stream. Dismantling consists of popping any modules on the Stream and closing the driver. Before a module is popped, the close may delay to allow any messages on the write message queue of the module to be drained by module processing. Similarly, before the driver is closed, the close may delay to allow any messages on the write message queue of the driver to be drained by driver processing. If O\_NDELAY (or O\_NONBLOCK) is clear, close will wait up to 15 seconds for each module to drain and up to 15 seconds for the driver to drain. If O\_NDELAY (or O\_NONBLOCK) is set, the pop is performed immediately and the driver is closed without delay messages can remain queued, for example, if flow control is inhibiting execution of the writer queue service procedure. When all modules are popped and any wait for the driver to drain is completed, the driver close routine is called. On return from the driver close, any messages left on the driver's queues are freed, and the queue and stdata structures are deallocated.



Note:- STREAMS frees only the messages contained on a message queue. Any message or data structures used internally by the driver or module must be freed by the driver or module close procedure.

Finally, the user's file table entry and the vnode are deallocated and the file is closed.

### Inserting Modules

An advantage of STREAMS over the traditional character I/O mechanism stems from the ability to insert various modules into a Stream to process and manipulate data that pass between a user process and the driver. In the example, the character conversion module is passed a command and a corresponding string of characters by the user. All data passing through the module are inspected for instances of characters in this string the operation identified by the command is performed on all matching character. The necessary declarations for this program are shown below:

```
*include <string.h>
#include <fcntl.h>
#include <sttropts.h>
#define BUFLLEN      1024
/*
 * These defines would typically be
 * found in a header file for the module
 */
#define XCASE      1. /*change alphabetic case of char*/
```

```

#define DELELETE      2. /* delete char */
#define DUPLICATE     3. /* duplicate char */
main()

```

```

    char buf (BUFLen);
    int fd, count;
    struct strioctl stroctl;

```

The first step is to establish a Stream to be communications driver and insert the character conversion module. The following sequence of system calls accomplishes this:

```

if ((fd = open ("/dev/comm/01", O_RDWR)) < 0)
    perror ("open failed");
    exit (1);
if (ioctl(fd, I_, "|chconv") < 0)
    perror ("ioctl I_PUSH failed");
    exit      (2);

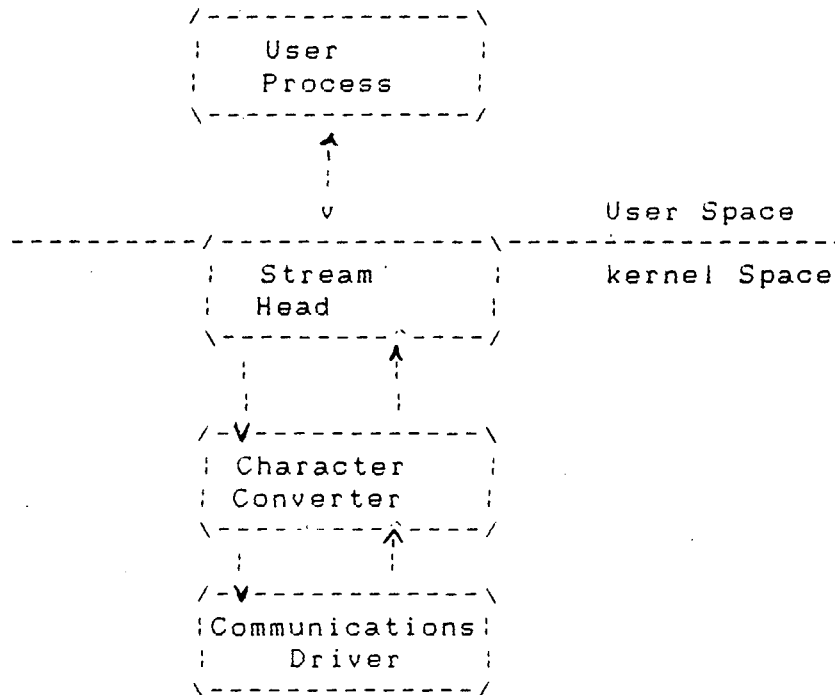
```

The I\_PUSH ioctl call directs the Stream head to insert the character conversion module between the driver and the Stream head, creating the Stream shown in Figure. As with drivers, this module resides in the kernel and must have been configured into the system before it was booted.

An important difference between STREAMS drivers and modules is illustrated here. Drivers are accessed through node or nodes in the file system and may be opened just like any other device. Modules, on the other hand, do not occupy a file system node. Instead, they are identified through a

separate naming convention, and are inserted into a Stream using I\_PUSH. The name of a module is defined by the module developer.

Figure: Case Converter Module



Modules are pushed onto a Stream and removed from a Stream in Last-In-First-Out (LIFO) order. Therefore, if a second module was pushed onto his Stream, it would be inserted between the Stream head and the character conversion module.

#### Module and Driver Control

The next step in this example is to pass the commands and corresponding strings to the character conversion module. This can be accomplished by issuing ioctl calls to the character conversion module as follows:

TH-4526



```

/* change all uppercase vowels to lowercase */
stroctl.ic_cmd - XCASE;
striocctl.ic_timeout - 0;          /* default timeout (15
sec) */
striocctl.ic_-- - "AEICU";
striocctl.ic_len - strlen (striocctl.ic_dp);
if (ioctl (fd, I_STR, striocctl) < 0)
    perror("ioctl I_STR failed").
    exit (3);

/* delete all instances of the chars 'x' and 'X' */
striocctl.ic_cmd - DELETE;
strriocctl.IC_dp - "xX";
striocctl.ic_len - strlen (striocctl.ic_dp)
If (ioctl (fd, I_STR, striocctl) < 0)
    perror ("ioctl I_STR failed:")
    exit (4).

```

ioctl requests are issued to STREAMS drivers and modules indirectly, using the I\_STR ioctl call. The argument to I\_STR must be a pointer to a striocctl structure, which specifies the request to be made to a module or driver. This structure is defined in <stropts.h> and has the following format:

Struct Striocctl

```

int ic_cmd: /* ioctl request */
int ic_timeout; /*ACK/NAK timeout */
int ic_len; /* length of data argument */
char *ic_dp; /* ptr to data argument */

```

Where `ic_cmd` identifies the command intended for a module or drivers `ic_timeout` specifies the number of seconds an `I_STR` request should wait for an acknowledgement before timing out, `ic_len` is the number of bytes of data to accompany the request, and `ic_dp` points to that data.

In the example two separate commands are sent to the character conversion module. The first sets `ic_cmd` to the command `XCASE` and sends as data the string "AEIOU"; it will convert all uppercase vowels in data passing through the module to lowercase. The second sets `ic_cmd` to the command `DELETE` and sends as data the string "xX"; it will delete all occurrences of the characters 'x' and 'X' from data passing through the module. For each command, the value of `ic_timeout` is set to zero, which specifies the system default timeout value of 15 seconds. The `ic_dp` field points to the beginning of the data for each command, the value of `ic_timeout` is set zero, which specifies the system default timeout value of 15 seconds. The `ic_dp` field points to the beginning of the data for each command. `ic_len` is set to the length of the data.

`I_STR` is intercepted by the Stream head, which packages it into a message, using information contained in the `strioc1` structure, and sends the message downstream. Any module that does not understand the command in `ic_cmd` will pass that message further downstream. The request will be processed by the module or driver closest to the stream head that understands the command specified by `ic_cmd`. The `ioctl` call will block up to `ic_timeout` seconds, waiting for the

target module or driver to respond with either a positive or negative acknowledgement message. If an acknowledgement is not received in `ic_timeout` seconds, the `ioctl` call will fail.

Note:- Only one `I_STR` request can retrieve the results, the any, of an `I_STR` request. If data are returned by the target module or driver, `ic_len` will be set on return to indicate the amount of data returned.

## CHAPTER IV

### DESCRIPTION & ALGORITHM FOR SIMPLE DRIVERS

#### **Loop-Around Driver**

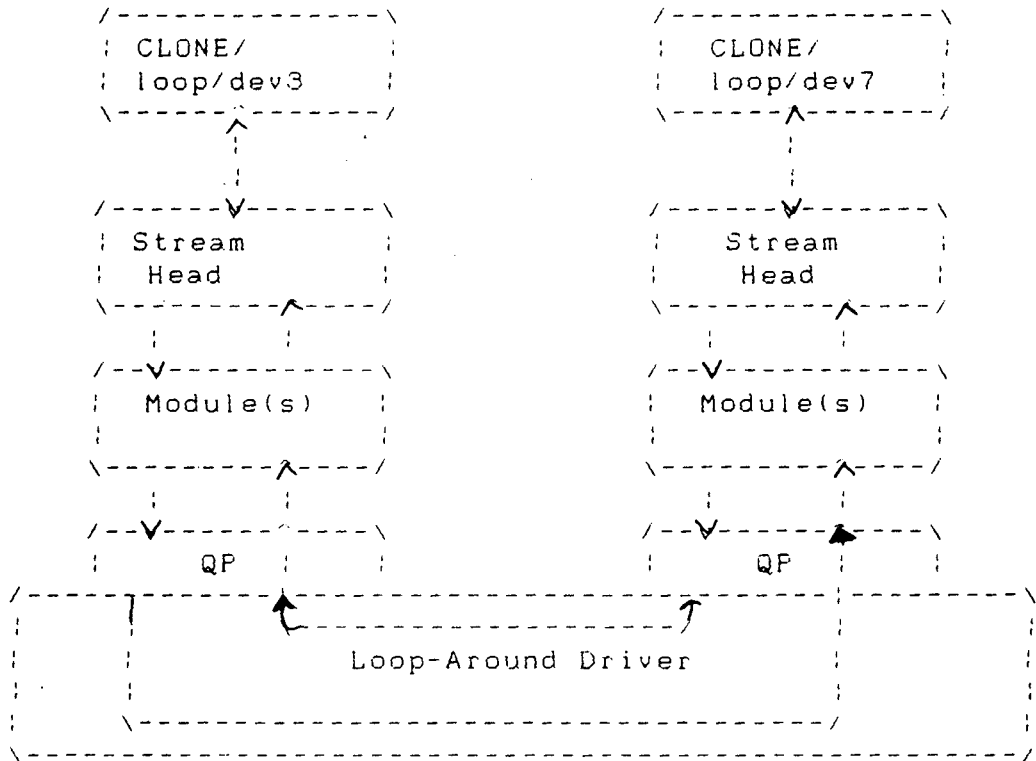
The loop-around driver is a pseudo driver that loops data from one open Stream to another open Stream. The user process see the associated files almost like a full-duplex pipe. The Streams are not physically linked. The driver is a simple multiplexer that passes messages from one Stream's write queue to the other Stream's read queue.

To create a connections, a process opens two Streams obtains the minor device number associated with one of the returned file descriptions, and sends the device number in an `I_STR` `ioctl` to the other Stream. For each open, the driver open places the passed queue pointer in a driver interconnection table, indexed by the device number. When the driver later receives the `I_IOCTL` message, it used the device number to locate the other Stream's interconnection table entry, and stores the appropriate **queue** pointer in both of the Streams' interconnection table entries.

Subsequently, when messages, other than `M_IOCTL` or `M_FLUSH` are received by the driver on either Stream's write-side, the messages are switched to the read queue following the drivers on the other Stream's read-side. The resultant logical connection is shown in Figure (in the figure, the abbreviation `QP` represents a queue pair). Flow control between the two Streams must be handled by special code

since STREAMS will not automatically propagate flow control information between two streams that are not physically interconnected.

Figure : Loop-Around Streams



The next example shows the loop-around driver code. The loop structure contains the interconnection information for a pair of Streams. `loop_loop` is indexed by the minor device number. When a Stream is opened to the driver, the address of the corresponding `loop_loop` element is placed in `q_ptr` (private, data structure pointer) of the read-side and write-side queues. Since STREAMS clears `q_ptr` `loop_loop` is used to verify that this Stream is connected to another open Stream.



The declarations for the driver are:

```
/*Loop_around driver */
#include "sys/types.h"
#include "sys/param.h"
#include "sys/sysmacros.h"
#ifdef u3b2
#include "sys/psw.h"
#include "sys/pcb.h"
#endif
#include "sys/stream.h"
#include "sys/stropts.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/cred.h"
#include "sys/ddi.h"
static struct module_info minfo ={
    Cxee 12,, "loop", 0, INFPSZ, 512, 128);
Static int loopopen (), loopclose(), loopwput(), loopwsrv():
static struct qinmit rinit ={
    NULL, looporrv, loopopen, loopclose, NULL, minfo, NUL,}
Static struct qinit winit ={
    loopwput, loopwsrv, NULL, NULL, NULL minfo, NULL);
Struct atreamtab loopinfo = { &init, &winit, NULL, NULL)
Struct loop{
queue loop{
```

```

queue_t *qptr; /*back pointer to write queue*/
queue_t *cqptr; /* pointer to connected read queue*/
#define LOOP_SET (('i <<8|1) /* should be in a file */
extern struct loop loop_loop():
extern int loop_cnt;
int loopdevflag - 0;

```

The open procedure includes canonical clone processing which enables a single file system node to yield a new minor device/vnode each time the driver is opened:

```

Static int loopopen (q, devp, flag, sflag, credp)
    queue_t *q;
    dev_t          *devp;
    int    flag;
    int    sflag;
    cred_t          *credp;
{
struct loop *loop;
dev_t newminor;
/*
*If CLONEOPEN, pick a minor device number to use.
* Otherwise , check the minor device range.
*/
If (sflag = CLONEOPEN) (
    for (newminor __ 0: newminor < loop_cnt; newminor ++){
        If (loop_loop [newminor] qptr -- NUL) break:
    }
}else
    newminor = getminor (* devp);

```

```

If (newsminor >= loop_cnt)
    return ENDCIO;

/* construct new device number and reset decp/*
/* getmajor gets the external major number, if (sflag =
CLONWOPEN)
*/
If (q->q_qtr) /*already open */
    return 0;

*devp = makedev (get major (*devp), newsminor);
loop&loop_loop (newsminor);
WR      (q)      ->q_qtr      =      (char*)      loop;
Loop. ptr = wr(q);
loop_qpnr -- NULL;
return 0;

```

In loopopen sflag can be CLONEOPEN, indicating that the driver should picture unused minor device (i.e., the user does not care which minor device is used In this case, the driver scans its private loop\_ data structure to find an unused minor device number. If sflag has not been set to CLONEOPEN, the passed-in minor device specified by getminor-> (\*devp) is used. Since the messages are awitched to the read queue following the other Stream's read-side, the driver needs a put procedure only on its write-side:

```

Static Int loopput (q, mp)
    queue_t *q;
    mblk_t *mp;

```

```

register struct, loop *loop;
loop = (struct loop *) q->q_qtr;
Switch (mp->b_bd_type);
case M_IOCTL; {
    Struct ioblk *iocp;
    int error;
    iocp = (struct iocblk *) mp->b_rptr;
    switch (iocp->ioc_cmd) {
case LOOP_SET; {
    int to; /* other minor device*/
    /* Sanity check, ioc_count contains the amount of
    * Sanity check, ioc_count contains the amount of
    * user supplied data which must equal the size of an
int.
    */
    If (iocp->ioc_count <= sizeof (int)).
    error = EINVAL::;
    goto iocnak;
}
    /* fetch other dev from @nd mesage block */
to = (int*)mp->b_rptr;
/* more sanity checks. The minor must be in range, open
already.
* Also, this device and the other one must be disconnected.
*/

```

```

If (to >-loop_loop (to) (to) qp) {
    error = ENAIO;
    goto iocnak;
}
If (loop_loop tr ll loop_loop (to), qp) {
    enmr = EBUSY;
    goto iocnak;
}

/* Cross connect Streams via the loop structures*/
loop->qp = RD(loop_loop(to).qp);
loop_loop [to], qp = RD (q);
/*
 * Return successful ioctl, Set ioc_count
 * to Zero, since no data are returned.
 */
mp->b_data->type = M_IOCACK;
ioc-> ioc_count = 0;
greply (q, mp);
break;
}

default:
    error = EINVAL;

iocnak;

/*
 *Bad ioctl, Setting ioc_error causes the
 *ioctl call to return that particular errno.
 * By default, ioctl will return EINVAL on failure

```

```

*/
mp->d_data->cd_type - M_IOCNAK;
iocp->ioc_error - error; /*ser returned arrno*/
}
break:
}

```

loopwput shows another use of an I\_STR ioctl call ("Module and Driver ioctls"). The driver sports a LOOP\_SET value of oic\_cmd in the iocblk of the M\_ioctl message. LOOP\_SET instructs the driver to connect the current open Stream to the Stream indicated in the message. The second block of the M\_IOCTL message holds an integer that specifies the minor device number of the Stream to connect to.

The driver performs several sanity checks: Does the second block have the proper amount of data? Is the "to" device in range? Is the "to" device open? Is the current Stream disconnected? Is the "to" Stream disconnected? If everything checks out, the read queue pointers for the two Streams are stored in the respective oqptr field. This cross-connects the two Streams indirectly, via loop\_loop.

Canonical flush handling is incorporated in the put procedure:

```

case                                                    M_FFLUSH:
    If (*mp->b_rptr - FLUSHW) {
        flushq (q, FLUSHALL); /* write */
        Flushq (loop=>optr, FLUSHALL);
        /* read on other side equals write on this

```

```

side*/
}
If (*mp->b_rptr : FLUSHR){
    flushq (RD (q), FLUSHALL);
    flushq (WR (loop_oqptr), FLUSHALL);
}
switch (*mp->b_rptr);
case FLUSHW:
    *mp->b_rptr - FLUSHW:
    break;
    case FLUSHR;
    *MP->b_rptr - FLUSHW:
    break;
)
    putnext (loop->oqptr. mp);
    break;
default: /*If this Stream ins't connected., send M_ERROR
upstream. */
    If (loop_>cqptr = NULL) (
        FREEMSQ (MP) ;
        PUTCTLL (RD) (q)->next, M_ERROR;ENXIO);
        break;
}
putq (q, mp);
}
}

```

Finally, loopwput enqueues all other messages e.g., M\_DATA or M\_PROTO) for processing by its service procedure. A check is made to see if the Stream is connected. If not, an M\_ERROR is sent upstream to the Stream head.

Certain messages types can be sent upstream by drivers and modules to the Stream head where they are translated into actions detectable by user process(es). The messages may also modify the state of the Stream head.

M\_ERROR Causes the Stream head to lock up. Message transmission between Stream and user processes is terminated. All subsequent system calls except close(2) and poll(2) will fail. Also causes an M\_FLUSH clearing all message queues to be sent downstream by the stream head.

M\_HANG UP Terminates input from a user process to the Stream. All subsequent system calls that would send messages downstream will fail. Once that would head read message queue is empty, EOF is returned on reads. Can also result in the SIGHUP signal being sent to the process group.

M\_SIG/M\_PCSIG Causes a specified signal to be sent to a process. putctl1() and putct() are utilities that allocate a non-data (i.e., not M\_DATA, M\_DELAY, M\_PROTO, or M\_PCPROTO) type message, place one byte in the message (for putctl(1)) and call the put procedure of the specified queues.

Service procedures are required in this example on both the write-side and read-side for flow control:

```
static int loopwsrv (q)
    register queue_t *p;
```



```

    mblkt *mp;
    register struct loop *loop;
    loop = (struct loop *) q->q_ptr;
    while (mp = getq (q) | = NULL) {
        /* Check if we can put the message up the other Stream
        read queue*/
        If (mp->b_datap->dbtype <- OPCIT " |canput (loop->oqptr-
        >q_next)
            break;
    }
    /* send message */
    putnext (loop->oqptr, mp); /* To queue following other
    Stream read queue*/
}
static int looprsrv (q)
    queue_t*q;
}
/* Enter only when "Back enabled" by flow control */
struct loop *loop:
loop-(struct loop*) q->q_ptr;
If (loop->oqptr = NULL)
return,
/* manually enable write service procedure */
qenable (WR (loop->cqptr));

```

The write, service procedure, loopwsrv takes on the canonical form. The queue being written to is not downstream but upstream (found via aqpir) on the other Stream.

In this case, there is no read-side put procedure so the read service procedure, `looprsrv`, is not scheduled by an associated put procedure, as has been done previously `looprsrv` is scheduled only by being back-enabled when its upstream becomes unstruck flow control blockage. The purpose of the procedure is to re-enable the write (`loopwsrv`) by using `oqpir` to find the related queue. `loopwsrv` can not be directly back-enabled by STREAMS because there is no direct queue linkage between the two Streams. Note the no message over gets queued to the read service, procedure, Messages are kept on the write-side so that flow control can propagate up to the Stream head. The `qenable ()` routine schedules the write-side service procedure of the Streams.

```
static int loopclose (q, flag, credp)
```

```
    queue_t *p)
```

```
    int      flag;
```

```
    cred_t  *credp;
```

```
    register struct loop *loop;
```

```
    loop = (struct loop*) q->q_ptr;
```

```
    loop->qptr = NULL;
```

```
/* If we are corrected to another stream break the
```

```
* linkage and send a hangup message.
```

```
* The hangup message causes the stream head to fall writes.
```

```
* allow the queued data to be read completely, and then.
```

```
* return EOF on subsequent reads.
```

```
*/
```

```
If (loop->qptr);
    (struct loop*) loop->q_ptr) ->qoptr = NULL;
    putctl (loop->q_next, M-HANGUP);
    loop->qptr - NULL;
    }
    }
```

loopclose sends an M\_HANGUP message up the connected Stream to the Stream head.

Note:- This drive can be implemented much more cleanly by actually linking the q-next pointers of the queue pairs of the two Streams.

## CHAPTER - 5

### Conclusions & Findings

A Driver is a software that provides an interface between the operating system and device. The driver controls the device in response to kernel commands and user-level programs access the device through system calls. The system calls interface with the file system and process control system, which in turn access the drivers. The drivers provides and manages a path for the data to and from the hardware device, and services interrupts issued by the device controller.

In general, drivers are grouped according to the type of the device they control, the access method (they way data are transferred), and the interface between the driver and the device. The type can be hardware or software. A hardware driver controls a physical device such as a disk. A software, driver, also called a pseudo device, controls software, which in turn may interface with a hardware device. The software driver may also support pseudo devices that have no associated physical device.

Drivers can be character-type or block-type but many support both access methods. In character-type transfer, data are read a character at a time or as a variable length stream of bytes, the size of which determined by the device. In block-type access, data transfer is performed on fixed-length block of data. Devices that support both block-and

character-type access must have a separate special device file for each access method. Character access devices can also use "raw" (also called unbuffered) data transfer that takes place directly between user address space and the device. Unbuffered data transfer is used mainly for administrative functions where the speed of the specific operation is more important than overall system performance.

The driver interface refers to the system stream----- and kernel interfaces used by the driver. For example, STREAMS in an interface.

### **Driver Configuration**

For a driver to be recognized as part of the system information on driver type, where object code resides, interrupts and seen must be stored in appropriate files.

The following summarizes information needed to include a driver in the system.

`/etc/mastered` This directory contains the master files. A master file supplies information to the system initialization software to describe different attributes of a drivers. There is one master file for each driver in the system.

`/stand/system` This file contains entry for each driver and indicates to the system initialization whether a driver is to be included or excluded during configuration.

/dev                    This directory contains  
                         provide applications with  
                         drivers via file operators

/boot                   This directory contains boot  
                         that are used to create a  
                         UNIX operating system when  
                         booted.

### **Writing a Driver**

All drivers are identified by a string character called the prefix. The prefix master file for the driver and is added driver routines. For example, the open driver with the "xyz" prefix is xyzopen.

The location of the driver source code whether the driver is a part of the core or an add-on to the core operating system.

Writing a driver differs from writing in the following ways:

- 1     A driver does not have a main C routine. Entry points are given specifically through switch tables.
- 2     A driver functions as a part of the kernel. In fact, a poorly written driver can degrade or corrupt the system.

- 3 A driver cannot use system calls or the C library, because the driver functions at a lower level.
- 4 A driver cannot use floating point arithmetic.
- 5 A driver cannot use archives or shared libraries, but frequently, used subroutines can be put in separate files in the source code directory for the driver.
- 6 Driver code, like other system software, uses the advanced C language capability. These includes bit-manipulation capabilities, casting of data, types, and use of header files for defining and declaring global data structures.
- 7 Driver code includes a set of entry point routines:- initialization entry points that are accessed through bdevsw | (block-access) and bdevsw (character-access) switch tables when the appropriate system call is issued.
- 8 Interrupt entry points that are accessed through the interrupt vector table when the hardware generates an interrupt.

The following lists rules of driver development:

- 1 All drivers must have an associated file in the master.d directory.
  - 2 All driver should have #include system header files that define data structures used in the driver.
  - 3 Drivers may have an init and / or a start routine to initialize the driver.
- Software drivers will usually have little to initial-

Software drivers will usually have little to initialize, because there is no hardware involved. An init routine is used when a driver needs to initialize but does not need any system services. init routines are run before system services are initialized (like the kernel memory allocator, for example). When a driver needs to do initialization that requires system services, a start routine is used. The start routines are run after system services have been initialized.

- Drivers will have open and close routines.
- Most drivers will have an interrupt handler routine.

The driver developer is responsible for supplying an interrupt routine for the device's driver. The UNIX system provides a few interrupt handling routines for hardware interrupts, but the developer, has to supply the specifics about the device.

In general, a prefixing interrupt routine should be written for any device that does not send separate transmit and receive interrupts. TTY devices that request separate transmit and receive interrupts can have two separate interrupt routines associated with them; prefix in it to transmit an interrupt, and prefix in it to receive an interrupt.

In addition, to hardware interrupts, many computers also support software interrupts. For example, AT&T computers support Programed Interrupt Request (PIRs). A PIR is



generated by writing an integer into a logical register address assigned to the interrupt vector table.

- Most drivers will have static subordinate driver routines to provide the functionality for the specific device. The names of these routines should include the driver prefix, although this is not absolutely required since the routine is declared as static.
- A bootable object file and special device files are also needed for a driver to be fully functional.

#### **Major and Minor Device Numbers**

The UNIX system V operating system identifies and accesses peripheral devices by major and minor numbers. When a driver is installed and a special device file is created, a device then appears to the user application as a file. A device is accessed by opening, reading, writing and closing a special device file that has the proper major and minor numbers.

The major number identifies a driver for a controller. The minor number identifies a specific device. Major numbers are assigned sequentially by either the system initialization software at boot time for hardware devices, by a program such as `drvinstall`, or by administrator direction. The major number for a software device is assigned automatically by the `drvinstall` command. Minor numbers are designated by the driver developer.

Major and Minor numbers can be external or internal.

External major numbers for software devices are static and assigned sequentially to the appropriate field in the master file by the `drvinstall(M)` command. External major numbers for hardware devices correspond to the board slot and are dynamically aligned by the `autoconfig` process at system boot time. The `mknod(IM)` command is then used to create the files (or nodes) to be associated with the device. External major numbers are those visible to the user.

Internal major numbers serve as an index into the `cdevsw` and `bdevsw` switch tables. These are assigned by the auto-configuration process when drivers are loaded and they may change every time a full-configuration boot is done. The system uses the `MAJOR` table to translate external major numbers to the internal major numbers needed to access the switch tables.

Minor numbers are determined differently for different types of devices. Typically, minor numbers are an encoding of information needed by the controller board.

External minor numbers are controlled by a driver developed by a driver developer, although there are conventions enforced for some types of devices by some utilities. For example, a tape driver may interface with a hardware controller (device) to which several tape driver (subdevices) are attached. All tape drives, attached to one

controller will have the same external major number, but each drive will have a different external minor number.

Internal minor numbers are used hardware drivers to identify the logical controller that is being addressed. Since drivers that control multiple devices (controllers) usually require a data structure for each configured device, drivers address the per-controller data structure by the internal minor number rather than the external major number.

The logical controller numbers are assigned sequentially by the central controller farmer at self-configuration time. The internal minor device number is calculated from the MINOR array in the kernel by multiplying the logical controller number by the value of the #DEV field (number of devices of per controller) in the master file.

The internal minor number for all software drivers is 0.

The MAJOR and MINOR tables map external major and minor numbers to the internal major number. The switch tables will have only as many entries as required to support the drivers installed on the system. Switch table entry points are activated by system calls that references a special device file that supplies the external major number and instructions on whether to use bdevsw or cdevsw. By mapping the external major number to the corresponding internal major number in the MAJOR table, the system knows which driver routine to activate. The routines getmajor() and getminor()

return an internal major and minor number of the device. The routines `getmajor ()` and `getminor()` return an external major and minor number for the device.

### **STREAMS Drivers**

At the interface to hardware devices, character I/O drivers have interrupt entry points' at the system interface, those same drivers, generally have direct entry points (routines) to process `open`, `close`, `read`, `write`, `poll`, and `ioctl` system calls.

STREAM device drivers have interrupts entry points at the hardware device interface and have direct entry point only for the `open` and `close` system. calls. These entry points are accessed via STREAMS, and the call formats differ from traditional character device drivers. (STREAMS drivers are character drivers, too. We call the non-STREAMS character drivers traditional character drivers or non-STREAMS character drivers). The `put` procedure is a driver's third entry point, but it is a message (not system) interface. The Stream head translates `write` and `ioctl` calls into messages and sends them downstream to be processed by the driver's write queue `put` procedure, `read` is seen directly only by the Stream head, which contains the functions required to process system calls. A driver does not know about system interface other than `open` and `close`, but it can detect the absence of a read indirectly if flow control propagates from the Stream head to the driver and affects the driver's ability to send messages upstream.

For input processing when the driver is ready to send data or other information to a user process, it does not wake up the process. It prepares a message and sends it to the read queue of the appropriate (minor device) Stream. The driver's write routine generally stores the queue address corresponding to this Stream.

For output processing, the driver receives messages in place of a write call. If the message can not be sent immediately to the hardware, it may be stored on the driver's write message queue. Subsequent output interrupts can remove messages from this queue.

Figure shows multiple Streams (corresponding to minor devices) to a common driver. There are two distinct Streams opened from the same major device. Consequently, they have the same streamtab and the same driver procedures.

The configuration mechanism distinguishes between STREAMS devices and traditional character devices, because system calls to STREAMS drivers are processed by STREAMS routines, not by the UNIX system routines. In the `cdevsw` file, the field `d_str` provides this distinction.

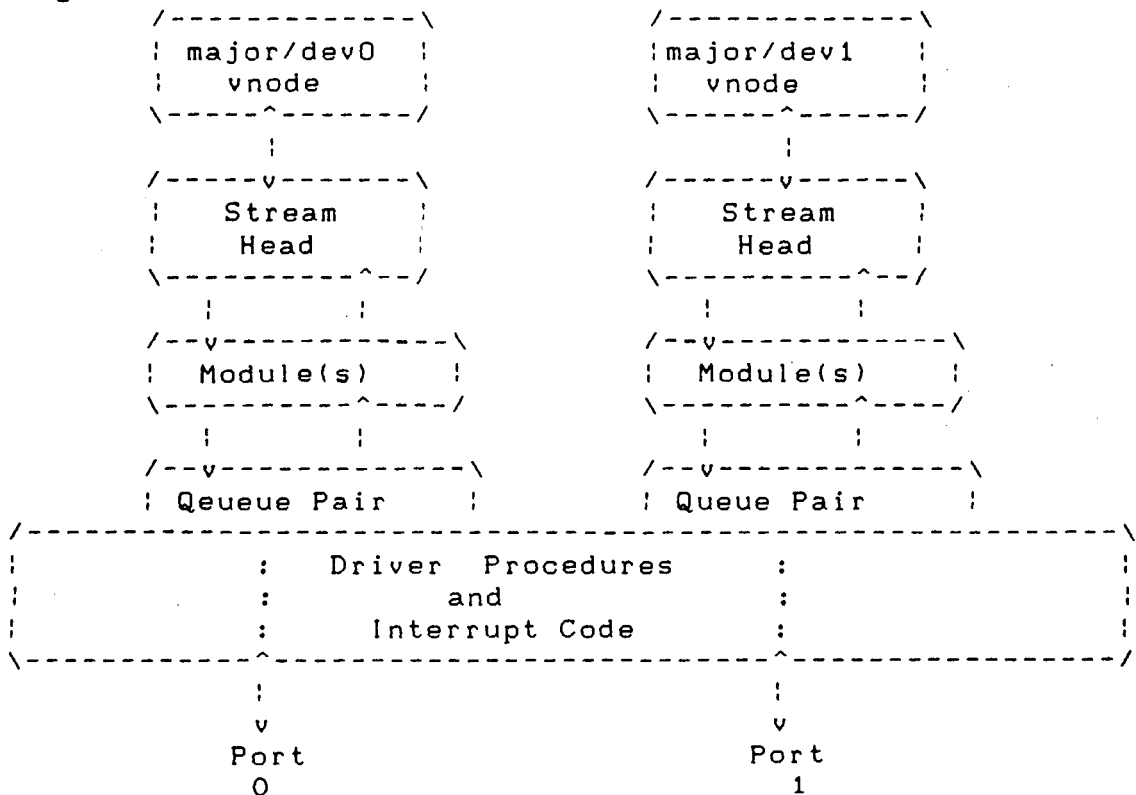
Multiple installation (minor devices) of the same driver are handled during the initial open for each device. Typically, the queue address is stored in driver-private structure array indexed by the minor device, number. This is for use by the interrupt routine which needs to translate

from device number for a particular Stream. The q\_pir of the queue will point to the private data structure, entry. When the messages are received by the queue, the calls to the driver put and service procedures pass the address of the queue, allowing the procedures to determine the associated device.

A driver is at the field end of a Stream. As a result, drivers must include standard processing for certain message types that a module might simply be able to pass to the next component.

During the open and close routine the kernel locks the device anode. Thus, only one open or close can be active at a time per major/minor device pair.

**Figure: Device Driver Streams**



```
/******  
LISTING OF MODULES REQUIRED  
******/
```

lopa.c page 1.

```
# include <sys/types.h>  
# include <sys/param.h>  
# include <sys/stream.h>  
# include <sys/streamcros.h>  
# ifdef u3b2  
# include <sys/psw.h>  
# include <sys/pcb.h>  
# include <sys/stropts.h>  
# include <sys/dir.h>  
# include <sys/signal.h>  
# include <sys/user.h>  
# include <sys/open.h>  
# include <sys/file.h>  
# include <sys/cred.h>  
# include <sys/ddi.h>  
# include <sys/errno.h>  
# include "def.h"  
  
static struct module_info minfo = {  
    0x0a, "lopa", 0, INFPSZ, 0, 0}  
  
int loapopen (), lopaclose();  
int lopawput();  
  
static struct qinit rinit = {  
    NULL, NULL, loapopen, lopaclose, NULL, &minfo, NULL};  
static struct qinit winit = {  
    lopawput, NULL, loapopen, lopaclose, NULL, &minfo, NULL};  
struct streamtab lopainfo = {&rinit, &winit, NULL, NULL};  
  
it lopadevflag = 0;  
  
int lopaopenvflag = 0;  
  
int lopaopen (q, devp, flag, sflag, credp)  
queue_t *q;  
dev_t *devp;  
int flag;  
cred_t *credp;  
{  
    return 0;  
}
```

```

int lopawput (q, mp)
  queue_t *q;
  mblk_t *mp;
{
  struct rst *ptr;
  mblk_t *bp;
  int i;

  ptr = (struct rst *) mp->b_rptr;
  /*
  if (ptr-> prim == CONNECT)

  ptr->prim = 4;
else
  ptr->prim = 5;
  */
  if (mp->b_cont != NULL)
  {
    bp = mp->b_cont;
    bcopy ("loops", bp->b_rptr, 7);
  }
#ifdef 0
  printf ("putting message from loopa driver\n");
#endif
  /*putnext (RD(q), mp);*/
  qreply (q,mp);
}
int lopaclose (q, flag, credp)
  queue_t *q;
  int flag;
  cred_t *credp;
{
  return 0;
}

```

mod.c page 1

```

# include <sys/types.h>
# include <sys/param.h>
# include <sys/stream.h>
# include <sys/stropts.h>
# include <sys/file.h>
# include <sys/open.h>
# include <sys/cred.h>

```



```

# include <sys/cmn_err.h>
# include <sys/ddi.h>
# include <sys/log.h>
# include <sys/strlog.h>

static struct module_info rminfo =
    {0x07, "mod", 0, INFPSZ, 512, 128};
static struct module_info wminfo =
    {0x07, "mod", 0, INFPSZ, 512, 128};
int modopen (), modwput (), modwsrv (), modrput ().
    modrsrv(), modclose();

int timhello();

static struct qinit rinit = {
    modrput, NULL, domopen, modclose, NULL, &rminfo, NULL};

static struct qinit winit = {
    modwput, dodwsrv, NULL, NULL, NULL, &wminfo, NULL};

Ostruct streamtab modinfo = { &rinit, &winit, NULL, NULL};

int moddevflag = 0, mod_tim_id;

int modopen(q, devp, flag, sflag, credp)

queue_t *p;
dev_t *devp;
int flag;
int sflag;
cred_t *credp;
{
#ifdef 0
    STRLOG(0x07,0,0,SL_TRACE,"inside mod open");
    mod_tim_im = timeout (timhello, 0, 1*HZ);
#endif

    return 0;
}

int modrput(q, mp)
    queue_t *q;
    mblk_t *mp;
{
#ifdef 0
    cmn_err(CE_CONT, "mod : put message, in modput\n");;
    cmn_err(CE_CONT, "modrput queue addr = %x\n", q)
    STRLOG(0x07,0,0,SL_TRACE, "mod :put message, in modput\n");

```

```

#endif
    switch(mp-> b_datap->db_type)
    {
        case M_FLUSH :
            if( *mp->b_sptr & FLUSHW)
                flushq ((q->q_flag & QREADR)? WR(q) : w, FLUSHDATA);
            break;

        default :
            putnext (q,mp);
    }
#if 0
    STRLOG (OX07,0,0,SL_TRACE, "mod : exiting modput\\n");
    cmn_err(CE_CONT, "modrupt queue addr after putnext = %x\\n". q)
#endif
}

int modwput (q, mp)
    queue_t *q;
    mblk_t *mp;
{
#if 0
    cmn_err(CE_CONT, "mod: wput /n");
    cmn_err(CE_CONT, "modqput queue addr = %x\\", q);
#endif
    strlog(Ox07,0,0,SL_ERROR, "mod : put message, in modput\\n");
    stwitch(mp-> b_datap->db_type)
    {
        case M_FLUSH :
            if(*mp->b_rptr & FLUSHW)
                flushq ((q->q_flag & QREADR) ? WR(q) : a, FLUSHDATA)
                break;

        default :
            /*
                putnext(q,mp);
            */
            /*
                putq(q,mp);
            */
    }
#if 0
    strlog (Ox07,0,0,SL_ERROR, "mod : exiting modput\\n");
    cmn_err(CE_CONT, "modwput queue addr after putq = %x'\\n", q);
#endif
}

int modwsrv(q)
    queue_t *q;
{

```

```

    mblk_t *mp;
#if 0
    cmn_err(CE_COUNT, "mod : wsrv\n");
    cmn_err(CE_CONT, "modwsrv queue addr = %x\n:", a);
#endif
    while ((mp = getq(q)) != NULL
        {
            if (cnaput (q->q_next))
            {
                # if 0
                cmn_err(CE_CONT, "modwsrv: putnext\n");
                #endif
                putnext (q, mp);
            }
            else
            {
                #if 0
                cmn_err(CE_CONT, "modwrv: putbq\n");
                #endif
                putbq(q, mp);
                return; /*qenable (q);*/
            }
        }
    #if 0
    cmn_err(CE_CONT, "modwsrv queue addr after while = %x\n", q)
    #endif
}

int modclose (q, flag, credp)
queue)t *q;
int flag;l
cred_t *credp;
{
#if 0
    strlog (0x07,0,0,SL_TRACE, "inside mod close");
    untimeout (mod_tim_id);
#endif
    return 0;
}

int timhello ()
{
# if 0

    cmn_err (CE_NOTE, "hello time o-- t\n");
#endif

```

```
}
```

```
moda.c page 1
```

```
# include <sys/types.h>
# include <sys/param.h>
# include <sys/stream.h>
# include <sys/stropts.h>
# include <sys/file.h>
# include <sys/open.h>
# include <sys/cred.h>
# include <sys/cmn_err.h>
# include <sys/ddi.h>
# include <sys/fcntl.h>

static struct module_info rminfo =
    {0x08, "moda", 0, INFPSZ, 0, 0};
static struct module_info wminfo =
    {0x08, "MODA", 0, INFPSZ, 0, 0};
int modaopen (), modaput (), modaclose ();

static struct qinit pinit = {
    modaput, NULL, modaopen, modacloee, NULL, &

static struct qinit winit = {
    modaput, NULL, NULL, NULL, NULL, & wminfo, NULL,};

struct streamtab modainfo = { & init, & winit, NULL, NULL};

int modadevflag = 0;

int modaopen (q, devp, flag, sflag, credp)
queue_t *q;
dev_t *devp;
int flag;
int sflag1;
cred_t *credp;
{
    return 0;
}

int modaput (w, mp)
queue_t *q;
mblk_t *mp;
{
/*
    cmn_err(CN_CONT, "mode : put message, in modaput/n");
*\
```

```

    putnext )q,mp);
    cmn_err (CE_CONT, "moda : end of modaput\n");
*\
}

```

```

int modaclose (q, flag, credp)
    queue_t *q;
    int flag;
    cred_t *credp;
{
    return 0;
}

```

modb.c page 1

```

# include <sys/types.h>
# include <sys/param.h>
# include <sys/stream.h>
# include <sys/stropts.h>
# include <sys/file.h>
# include <sys/open.h>
# include <sys/cred.h>
# include <sys/cmn_err.h>
# include <sys/ddi.h>

```

```

static struct module_info rminfo = {0x08, "MOBD", 0,0,0, 0};
static struct module_info wminfo = {0x08, "modb", 0,0,0, 0};
int modbopen (), modbput (), mobdclose ();

```

```

static struct qunit rrinit = {
    modbput, NULL, NULL, NULL, NULL, &winit, NULL};

```

```

static struct qinit winit = {
    modbput, NULL, NULL, NULL, NULL, &wminfo, NULL};

```

```

int modbdevflag = 0;

```

```

int modbopen(q, devp, flag, sflag, credp)
    queue_t (*q;
    int flag;
    int sflag;

```

```

    cred_t *ccredp;
{
    return 0;
}

```

```

int modbput)q, mp)

```

```

queue_t *q;
mblk_t *mp;
{
/*
    cmn_err(CE_CONT, "modb : put message, in modbput\n");
*/
    switch(mp-> b_datap->db_type)
    {
case M_FLUSH :
        if (*mp->b_optr & FLUSHW)
            flushq ((q->q_flag & QREADR) ? WR (a) : q, FLUSHDTA)
            break;

default :
            putnext (q,mp);
    }
}
int modbclose(q, flag, credp)
queue_t *q;
int flag;
cred_t *credip;

{
    return 0;
}

```

usrstr.c page 1

```

# include <sys/stdio.h>
# include <sys/stropts.h>
# include <sys/poll.h>
# include <sys/fcntl.h>
# include <sys/errno.h>
# include <sys/time.h>
# include "def.h"

#define NPOLL 2

main ()
{
FILE *fp, *ofp[2];
int val, flgs=0;
int prim_typ, cnt=0;
char datbuf[1000], s[100];
struct ltm sndtm, rcvtm;
struct strioctl stio;

```

```

struct strbuf ctlstr, datstr;
struct pollfd pollfds[NPOLL];
int i;

if ((pollfds[0].fd=open ("/dev/str2", O_RDWR)) < 0)
{
    printf("fd0 : 0 : open failed\n");
    printf("error = %d\n", errno);
    exit(0);
}
if (ioctl(pollfds [0].fd, I_PUSH, "mod") < 0)
{
    Printf("fd0 : I_PUSH failed\n");
    Printf("error = %d\n", erre);
    exit(0);
}
if ((pollfds[1].fd = open ("/dev/str2", O_RDWR)) < 0)
{
    Printf("fd1 : open failed\n");
    printf("error = %d\n", errno);
    exit (0);
}

if (ioctl(pollfds[1].fd, I_PUSH, "mod") < 0)
{
    printf("fd1 : I_PUSH failed\n");
    printf("error = %d\n, arrno);
    exit(0);
}

stio.ic_cmd = 1;
stio.ic_timeout = 0;
stio.ic_len = 0;
stio.ic_dp = NULL;
if (ioctl (pollfds[0].fd. I_STR, &stio)< 0)
{
    printf("I_STR failed\n");
    printf("error = %d\n", errno);
    exit (0);
}
if ((fp = fopen ("tstdatt",, "n")) == NULL)
{
    printf("tstdat : fopen failed\n");
    printf("error = %d\n", errno);
    exit(0);
}

```

```

if ((ofp[0] = fopen ("ourdat", "w")) == NULL)
{
    printf("outdat : fopen failed\n");
    printf("error = %d\n", errno);
    exit(0);
}

if ((ofp[1] = fopen ("ourdat1", "w")) == NULL)
{
    printf("outdat1 : fopen failed\n");
    printf("error = %d\n", errno);
    exit(1);
}

pollfds[0].events = POLLIN;
pollfds[1].events = POLLIN;

ctlstr.maxlen = sizeof(int);
ctlstr.len     = sizeof(int);
stlstr.buf     = (char *) & jprim_typ;

for (i=0 i<996; i++)
    datbuf[i] = 'a';
datbuf[996] = '\0';

while(1)
{
    if (fgets(s, 100, fp) == NULL)
    {
        printf("invalid data\n");
        exit(0);
    }
    /*printf("%s\n", s);
    sscanf (s, "%d %s", &prim_type, datbuf);
*/
    sscanf (s, "%d", %s", &prim_type);
    /*printf("prim = %d dat = %s\n", prim_typ, datbuf);*/

    if (prim_typ == CLOSS)
    {
        gettimeofday(&rcvtime);
        break;
    }

    datstr.maxlen = sizeof(datbuf);
    datstr.len     = strlen(datbuf) + 1;
    dtstr.buf      = datbuf;

```



```

if (prim_typ == CONNECT) gettime (&sndtm);;
cnt++;
if ((val=putmsg (pollfds[0], fd, &ctlste, &datstr, flgs,)) < 0)
{
    printf("putmsg failed\n");
    exit (0);
}
/* printf("going to poll msg # %d return
           val of putmsg = %d \n", cnt, val);*/

    if (poll(pollfds, NPOLL, -1) <)
{

    printf("poll failed\n");
    exit(0);
}
printf("polled msg # %d\n", cnt);
    for (i=0; i < NPOLL; i++)
{
switch (pollfds[i]. revents)
{
    case POLLIN :
        datstr.mazlen = sizeof (struct rst);
        datstr.len     = 0;
        datstr.buf     = (char *) &rcvstr;

        if ((val = gatmsg(pollfds[i]., fd, &ctlstr, &datstr, &flgs)) <
            {
                printf("getmsg failed\n");
                exit(0);
            }

            sndtm.mm, sndtm.ss);

        printf("%d\n", *((int *)ctlstr.buf));
        printf(ofp[i], "%d %s\n", *((int *)ctlstr.buf), rcvstr.dat);
/*

        printf("%d %s\n", *((int *)ctlstr.buf), ctlstr.buf, rcvstr
        fprintf(ofp, "receive time = %d %d %d\n", rcvtm.hh,
            rcvtm.mm, rcvtm.ss);
*/

        break;

    case 0:
        printf("No Event \n");
        break;
}
}
}

```

```

        default :
                printf("error event \n");
                exit (0);
        }          /* end of switch */
}          /* end of for loop */

}          /* end of while loop */
for (i=0; i<NPOLL; i++)
{
printf(ofp[i], "send time = %d %d %d/n", sndtm.hh, sndtm.mm,
                snotm.ss);
printf (ofp[i], "receive time = %d %d %d\n", revtm.hh,
                rcvtm.mm, rrcvtm.ss);
printf(ofp[i], "#of msgs = %dn", cnt);
close(jpollfds[[i].fd);
close(ofp[i]);
}
        close(fp);
}

gettime(ptm)
struct ltm *ptm:
{
        time_t tim;
        struct tm * localtime();
        struct tm *ttm;

        time (&tim);
        ttm = localtime(&tim);
        ptm->hh = ttm->tm_hour;
        ptm->mm = ttm->tm_min;
        ptm->ss = ttm->tm_sec;
}

```

simres.c page 1

```

# include <sys/stdio.h>
# include <sys/stropts.h>
# include <sys/poll.h>
# include <sys/fcntl.h>
# include <sys/errno.h>
# include <sys/time.h>
# include "def.h"

#define NPOLL 1

main ()

```

```

{
FILE *fp, *ofp;
int fd0, fd1, fd2, flgs=0;
int prim_typ, cnt=0;
char datbuf[1000], s[210];
struct ltm sndtm, rcvtm;
struct rst rcvstr;
struct strbuf ctlstr, datstr;
struct pollfd pollfds[NPOLL];
int i;
if (( fd1 = open ("/dev/res",o_rdwr))<0)
{
printf("open failed\n");
printf("error= %d\n,errno);
exit(0);
}
if((fd0 = open("/dev/res", O_RDWR)) < 0)
{
printf("oen ("open failed\n");
printf("error = %d\n", errno);
exit(0);
}
if ((fd0= open ("/dev/rmux", O_RDWR)) < 0)
{
printf("open failed\n");
printf("error = %d\n", errno);
exit(0);
}
if (ioctl(fdo, I_LINK, fd1) < 0)
{
printf("res: I_LINK failed\n"); failed\n");
printf("error = %d\n", errno);
exit(0);
}
if
((fd2 = open ("/dev/lope", O_RDWR)) < 0)
{
printf("lopa : I_LINK, failed\n");
printf("error = %d\n", errno);
exit(0);
}

close(fd1);
close(fd2);

if (ioctl(fdo, I_PUSH, "mod") < 0)
{

```

```

    pprintf("I_PUSH, failed\n");
    printf("error = %d\n", errno);
    exit(0);
}
if printf("I_PUSH, "modc") < 0)
{
    printf("I_PUSH failed\n");
    printf("error = %d\n", errno);
    exit(0);
}
if (ioctl(fd0, I_PUSH, "modb") < 0)
{
    printf("I_PUSH failed\n", errno);
    exit(0);
}

if (ioctl(fd0, I_PUSH, "moda") < 0)
{
    printf("I_PUSH, failed\n");
    printf("error = %d\n", errno);
    exit(0);
}
/*****

if ((fd1= open ("dev/umux", O_RDWR)) < 0)
{
    printf("umux : open failed\n");
    printf("error = %d\n", errno);
    exit(0);
}
printf("rmux: I_LINK, fd0) < 0)
{
    printf("rumx : I_LINK failed\n");
    printf("error = %d\n", errno);
    exit(0);
}
close(fd0);
/*****

if ((fp = fopen ("tstdat", "n")) == NULL
{
    printf("fopen failed\n");
    exit(0);
}

pollfds[0].fd = fd1;
pollfds[0].events = POLLIN;

```

```

ctlstr.maxlen = sizeof(int);
ctlstr.len    = sizeof(int);
ctlstr.buf    = (char *)&prim_typ;

for (i=0; i<996; i++)
datbuf[[i] = 'a';
datbuf[996] = '\0';

printf("press enter\n");
getchar();

while(1)
{
if (fgets(s, 120, fp) == NULL)
{
printf("invalid data\n");
}
/*
scanf (s, "%d %s", &prim_typ);
if (prim_typ == CLOSE)
{
gettime (&rcvtm);
break:
}
datstr.maxlen = sizeof(datbuf);
datstr.len    = strlen(datbuf); + 1;
datstr.buf    = datbuf;
    of (cnt == 0) gettime(&sndtm);

cnt++;
/*
if (cnt == 500)
{
printf("press return to continue\n");
getchar();
}
*/
if (putmsg (gdl, &ctlstr, &datstr, flgs) < 0)
{
printf("putmsg failed\n");

exit (0);
}
if (poll(pollfds, NPOLL, -1) < 0)
{
printf("poll failed \n");
exit(0);
}

```

```

    }
    for (i=0;i < NPOLL; i++)
    exit(0);
    }
    for (i=0; i < NPOLL; i++);
    {
    switch (pollfds[i]. revents)
    {
    case POLLIN:
        datstr.maxlen = sizeof (struct rst);
        datstr.len     = 0;
        datstr.buf     = (char *) &rcvstr;

        if (getmsg(fd1, &ctlstr, &datstr, &flgs) < 0)
        {
            printf("getsg failed\n");
            exit(0);
        }
        fprintf(ofp, "%d %s\n", *((int *) ctlstr.buf), rcvstr.dat);
        break;

    case 0 ;
        printf("No" Event\n");
        break;
    default :
        printf("error event \n");
        exit(0);
    } /* end of switch */
    } /* end of for loop */
    } /* end of while loop */

    fprintf(ofp, "send time = %d %d %d/n", sndtm.hh, sndtm.mm,
    sndtm.ss);
    fprintf(ofp, "receive time = %d %d %d\n", rcvtm.hh, rcvtm.mm,
    rcvtm.ss);
    printf("press return to exit\n");
    getchar();

    if (ioctl(fd1, I_UNLINK, -1 ) < 0)
    {
        printf("umux: I_UNLINK failed\n");
        printf("error = %d\n", errno);
        exit(0);
        close(fd1);
        fclose(fp);
        fclose(ofp);
    }

```

```

gettime(ptm)
struct ltm *ptm;
{

time_t tim;
struct tm *localtime();
struct tm *ttm;

time(&time);
tmm, = localtime(&tim);
ptm->hh - ttm->tm_hour;
ptm->mm - ttm->tm_min;
ptm->ss - ttm->tm_sec;
}

# include <sys/types.h>
# include <sys/param.h>
# include <sys/stream.h>
# include <sys/sysmacros.h>
# ifdef u3b2

# include <sys/psw.h>
# include <sys/pcb.h>
# endif
# include <sys/stropts.h>
# include <sys/dir.h>
# include <sys/signal.h>
# include <sys/user.h>
# include <sys/open.h>
# include <sys/file.h>
# include <sys/cred.h>
# include <sys/ddi.h>
# include <sys/errno.h>
# include <sys/cmn_err.h>
/*#include <time.h>*/
#include "def.h"

static struct moduleinfo imnfo = {
0x09, "rres", 0, INFPSZ, 0, 0 };

int resopen(), resclose();
int reswput();

static struct qinit rinit = {
NULL, NULL, resopen, resclose, NULL, &minfo, NULL};

```

```

static struct qint qinit = {
    reswput, NULL, NULL, NULL, NULL, &minfo, NULL};

struct streamtab resinfo = { &rinit, &winit, NULL, NULL};

int resdevflag = 0;

int resopen(q, devp, flag, sflag, credp)
queue_t *q;
dev_t *devp;
int flag;
int sflag;
cred)t *credp;
{
/*
    cmn_err(CE_CONT, "returning from resopen \n");
*/
    return 0;
}

int reswput(q, mp)
queue_t *q;
mblk_t *mp;
{
    struct rst *ptr;
    mblk_t *bp;

Jun    res.c

int i:L
ptr = (struct rst *) mp->b_rptr;
/*
if (ptr->prim == CONNECT)
{
    ptr->prim = 6;
printf('from minor device 0\n");
}
else
{
    ptr->prim = 9;
printf("from minor device 1\n");
}
*/

if (mp->b_cont != NULL)
{
    bp= mp->b_cont;

```



```

        bcopy("driver1", bp->b_rptp, 7);
    }
    /*putnext(RD(q), mp);
        qreply(q,mp);

    /*
    cmn_err(CE_CONT, "retuning from res put\n");
    */
}

```

```

int resclose (q, flag, credp)
queue_t *q;
int flag;
cred_t *credp;
{
    return 0;
}

```

rmux.c page 1

```

# include <sys/types.h>
# include <sys/param.h>
# include <sys/stream.h>
# include <sys/sysmacros.h>
# ifdef u2b2
# include <sys/psw.h>
# include <sys/pcb.h>
#endif
# include <sys/stropts.h>
# include <sys/dir.h>
# include <sys/signal.h>
# include <sys/user.h>
# include <sys/unistd.h>
# include <sys/file.h>
# include <sys/cred.h>
# include <sys/ddi.h>
# include <sys/errno.h>
# include "def.h"

```

```

static struct module info minfo = {
0x0c, "rmux", 0, INFPSZ, 0, } };

```

```

int rmuxopen(), rmuxclose();
int rmuxwput(), rmuxrput();

```

```

static struct qinit uwinit = {
    NULL, NULL, rmuxopen, rmuxclose, NULL, &minfo, NULL};

```

```

static struct qinit uwinit = {
    rmuxrput, NULL, NULL, NULL, NULL, &minfo, NULL};

static struct qinit irinit = {
    rmuxrput, NULL, NULL, NULL, NULL, &minfol, NULL};

struct treamtab rmuxinfo = {urinit, &uwinit, &lrinit, &lwinit};

int rmuxdevflag = 0;

struct linkblk lnkp[2], *lptr;

int rmuxopen(q, devp, flag, sflag, credp)
queue_t *q;
dev_t *devp;
int flag;

int sflag;

cred_t *credp;
{
    return 0;
}

int rmuxrput(q, mp)
queue_t *q;
mblk_t *mp;

{
    if(mp->b)rptr == CONNECT)
        putnext(RD(lnkp[0].l_qtop), mp);
    else
        putnext(RD(lnkp[1].I_qtop), mp);
}

int rmuxwput(q, mp)
queue_t *q;
mblk_t *mp;
{
    switch (mp->b_datap->db_type)
    {
    case M_IOCTL :
        {
            struct iocblk *iocp;
            iocp = (struct iocbl< *) mp->b)rptr;
            swtich (iocp->ioc cmd)
            {

```

```

case I_LINK :
    lptr = (struct linkblk *) mp->b_cont->b_rptr;
    if (lnkp[0].l_qtop = NULL)
    {
        lnkp[0].l_qtop = lptr->l_qtop;
        lnkp[0].l_qbot = lptr->l_qbot;
        lnkp[0].l_index = lptr->l_index;
    }
    else
    {
        lnkp[1].l_qtop = lptr->l_qtop;
        lnkp[1].l_qbot = lptr->l_qbot;
        lnkp[1].l_index = lptr->l_index;
    }
    mp->b_datap->db_type = M_IOCACK;
    iocp->ioc_count = 0;
    greply(q, mp);
    break;
case I_UNLINK :
    lptr = (struct linkblk *) mp->b_cont->b_rptr;
    if (lnkp[0].l_qbot != NULL)
    {
        lnkp[0].l_qtop = NULL;
        lnkp[0].l_qbot = NULL;
        lnkp[0].l_index = 0;
    }
    else
    {
        lnkp[1].l_qtop = NULL;
        lnkp[1].l_qbot = NULL;
        lnkp[1].l_index = 0;
    }
    mp->b_datap->db_type = M_IOCACK;
    iocp->ioc_count = 0;
    greply(q, mp);
    break;

    } /* end of ioctl switch */
    break;
}
default :
{
    int *mtyp;;

    mtyp = (int *) mp->b_rotr;
    if (*mtyp == CONNECT )

```

```
        putnext(linkp[0].l_qbot, mp);
    else
        putnext(lnkp[1].l_qbot, mp);

    break;
}
} /* end of switch */
}
int r muxclose(q, flag, credp)
queue_t *q;
int flag;
cred_t *credp;
{
    return 0;
}
```

## BIBLIOGRAPHY

1. [Bach 84] Bach, M. J. and S.J. Boriff, "Multiprocessor Unix System," A T & T Bell Laboratories Technical Journal, Oct. 1984, Vol 63, No. 8, Part 2, pp 1733-1750.
2. [Barak 80] Barak, A. B. and A. Shapir, "UNIX with Satellite Processors," Software-Practice and Experience, Vol. 10, 1980, pp. 383-392.
3. [Berkeley 83] UNIX Programmer manual, 4.2 Berkeley Software Distribution, Virtual Vax-11 Version Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, August 1983.
4. [Bodenstab 84] Bodenstab, D.E., T.F. Houghton, K. A. Kellman, G. Ronkin, and E.P. Schan, UNIX Operating System Porting Experiences, A T&T Bell Laboratories Technical Journal, Vol. 63, No 8, Oct 1984, pp.1769-1790.
5. [Bourne 78] Bourne, S.R., "The Unix Shell," The Bell System Technical Journal, July-August 1978, Vol. 57, No. 6, part 2, pp. 1971-1990.
6. [Cole 85] Cole, C.T., P.B. Flinn, and A.B. Atlas, "An implementation of an Extended File System For Unix Operating System," Proceedings of the USENIX Conference, Summer 1985, pp. 131-149.
7. [Dijkstra 68] Dijkstra, E.W. "Cooperating Sequential

- Process," in Programming Languages, ed. F. Genuys, Academic press, New York, NY, 1968.
8. [HOLT 83] Holt, R.C. Concurrent Euclid The Unix Operating System and Tunix, Addison-Wesely, Reading, MA, 1983.
  9. [Kerningham 78] Kerningham, B.W. and D.M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1978.
  10. [Kerningham 84] Kerningham, B.W. and D.M. Ritchie, The C Programming Language, Prentice Hall, Englewood Cliffs, NJ, 1978.
  11. [Killian 84] Killian T.J. "Processes as a file," Proceedings of the USENIX Conference. Summer 1984, pp 203-207.
  12. [Peachy 84] Peachy, D.R. R.B. Bunt, C.L. Williamson, and T.B. Brecht, "An Experimental Investigation of Scheduling Strategies for Unix Operating System" Performance Evaluation Review 1984 SIGMETRICS CONFERENCE on Measurement and Evaluation of Computer Systems, Vol 12(3), August 1984, pp. 158-166.
  13. [Postel 80] Postel, J., (ed), "DOS Standard Transmission Control Protocol," ACM Computer Communication Review, Vol. 10, No. 4, Oct 1980, pp. 52-132.
  14. [Richards 69] Richards, M. "BCPL: A Tool for Compiler

Writing And Systems Programmings, "Proc. AFIPS JCC 34 1969, pp. 557-566.

15 . [Thompson 74] Thompson, K., "UNIX Implementation," The Bell laboratories Technical Journal vol 63, No. 6, Part 2, October 1984, pp. 1815-1826.

16. [Weinberg 84] Weinberg, P.J., "Cheap Dynamic Instruction Counting," The AT&T BELL LAB Technical Journal