ATM Networking in Linux

Bindings occur at four distinct times:

- System initialization
- Device driver initialization
- Socket creation
- Socket connection

System initialization time:

Protocols known to Linux are declared in a static table in module protocols.c

```c
struct net_proto protocols[] = {
    { "UNIX",     unix_proto_init   },
    { "802.2",    p8022_proto_init },
    { "802.2TR",  p8022tr_proto_init },
    { "SNAP",     snap_proto_init } ,
    { "RIF",      rif_init    },
    { "AX.25",    ax25_proto_init },
    { "NET/ROM",  nr_proto_init } ,
    { "INET",     inet_proto_init },
    { "IPX",      ipx_proto_init },
    { "DDF",      atalk_proto_init },
    { "ATMPVC",   atmpvc_proto_init },
    { "ATMSVC",   atmsvc_proto_init },
    { NULL,       NULL     }
};
```

Protocol initialization modules are called in Linux kernel initialization.
ATMPVC initialization: `atmpvc_proto_init`

Its function is to register the protocol using the following call:

```c
sock_register(pvc_proto_ops.family,
            &pvc_proto_ops);
```

Family is `PF_ATMPVC` (as in `PF_INET`)

`pvc_proto_ops` is a table of entry point addresses:

```c
struct proto_ops {
    int family;
    int (*create) (struct socket *sock, int protocol);
    int (*dup)      (struct socket *newsock, struct socket *old);
    int (*release)  (struct socket *sock, struct socket *peer);
    int (*bind)     (struct socket *sock, struct sockaddr *myad,
                     int sockaddr_len);
    ...
};
```

For ATM PVCs the structure is filled in as follows:

```c
static struct proto_ops pvc_proto_ops = {
    PF_ATMPVC,
    atm_create,
    pvc_dup,
    atm_release,
    pvc_bind,
    pvc_connect,
};
```

- The entry point addresses are positionally standard for all protocols
- `sock_register()` saves the address of each protocol’s `proto_ops` table in the `pops[]` table
- The `pops[]` table is indexed by `PF_number`
After protocol initialization

```
struct proto_ops* 
pops[] = { 
    : 
    [PF_ATMPVC] 
```

```
struct proto_ops{
    PF_ATMPVC, 
atm_create, 
pvc_dup, 
atm_release, 
pvc_bind, 
pvc_connect, 
NULL, 
pvc_accept
```

- Generic Linux Networking
- Linux ATM Networking
- APE 25 Device Driver
Device driver initialization time:

From `init_module()` the driver must call

```c
struct atm_dev *atm_dev_register(char *type,
                                 struct atmdev_ops *ops,
                                 unsigned long flags)
```

The first parameter is the device name .... “ape25”
The second is a pointer to the device driver operations vector table:

```c
static struct atmdev_ops atm_ops =
{
  ape25_open,           /* open         */
  ape25_close,          /* close        */
  ape25_ioctl,          /* ioctl        */
  ape25_getsockopt,     /* getsockopt   */
  ape25_setsockopt,     /* setsockopt   */
  ape25_send,           /* send         */
  ...
};
```

`atm_dev_register` allocates a slot in its ATM Device Table
Device table entries are structures of the following content:

```c
struct atm_dev {
  const struct atmdev_ops *ops; /* device operations; */
  const struct atmphy_ops *phy; /* PHY operations, */
  const char  *type;            /* device type name */
  int      number;              /* device index */
  struct atm_vcc *vccs;         /* VCC table (or NULL) */
  struct atm_vcc *last;         /* last VCC (or undefined) */
  void     *dev_data;           /* per-device data */
  void     *phy_data;           /* private PHY date */
};
```

A pointer to the `atm_dev` structure is returned to the driver, and the
drivers “ape” structure and the `atm_dev` structure are linked.

```c
ape->atmdev = atm_dev_register(.....);
ape->atmdev->dev_data = ape;
```
After Device Driver Initialization

```c
struct atm_dev
  *vccs
  *atm_devops
  *dev_data

struct atm_devops =
  { ape25_open,
    ape25_close,
    : ape25_send

struct proto_ops{ PF_ATMPVC,
  atm_create,
  pvc_dup,
  atm_release,
  pvc_bind,
  pvc_connect,
  NULL,
  pvc_accept

struct pddtype ape
  { pci data
    buffer pool data
    performance data
    atm_device_data
```
Socket Creation Time

A socket is created with the usual call:

```c
socket(int family, int type, int protocol)
```

For example:

```c
s = socket(PF_ATMPVC, SOCK_DGRAM, ATM_AAL5);
```

A new socket structure is allocated:

```c
struct socket {
    short              type;    /* SOCK_STREAM, ...            */
    socket_state       state;
    long               flags;
    struct proto_ops   *ops;   /* protocols do most everything */
    void               *data;  /* protocol data (-> atmvcc) */
    struct socket      *conn;  /* server socket connected to */
    struct socket      *iconn; /* incomplete client conn.s */
    struct socket      *next;
    struct wait_queue  **wait; /* ptr to place to wait on */
    struct inode       *inode;
    struct fasync_struct *fasync_list; /* Asynch wake up list */
    struct file       *file;   /* File back pointer for gc */
}
```

The ATM protocol operations table pointer retrieved from `pops[PF_ATMPVC]` and stored in the socket structure

The protocol specific (`atm_create`) create function is then called.
ATM Socket Creation

`atm_create` allocates a VCC structure

```c
struct atm_vcc {
    unsigned short  flags;      /* VCC flags (ATM_VF_*) */
    unsigned char   family;     /* address family; 0 if unused */
    unsigned char   aal;        /* ATM Adaption Layer */
    short           vpi;        /* VPI and VCI (types must be */
    int             vci;        /* equal with sockaddr */
    unsigned long   aal_options; /* AAL layer options */
    unsigned long   atm_options; /* ATM layer options */
    struct atm_dev  *dev;       /* device back pointer */
    struct atm_qos  qos;        /* QOS */
    atomic_t         tx_inuse,rx_inuse; /* buffer space in use */
    void (*)push(struct atm_vcc *vcc,struct sk_buff *skb);
    void (*)pop(struct atm_vcc *vcc,struct sk_buff *skb);
    void            *dev_data;  /* per-device data */
    void            *proto_data; /* per-protocol data */
    struct timeval  timestamp; /* AAL timestamps */
    struct sk_buff_head recvq;  /* receive queue */
    struct socket   *sock;      /* Back pointer to our socket */
};
```

`atm_create` then

- Cross links the `vcc` and socket structures using the `*sock` field in the `vcc` and the `*data` field in the socket.
- Initializes some required fields in the `vcc`. 
After socket creation

- **Generic Linux Networking**
- **Linux ATM Networking**
- **APE 25 Device Driver**

### ATM Device Table

- `interface #`
- `struct atm_dev *vccs`
- `*atm_devops`
- `*dev_data`

### Sockets

- `struct socket`:
  - `*proto_ops`
  - `*data`

### ATM Device Operations

- `struct atm_devops = {`
  - `ape25_open`,
  - `ape25_close`,
  - `ape25_send`

### PDD Types

- `struct pddtype ape`:
  - `pci data`
  - `buffer pool data`
  - `performance data`
  - `atm_device_data`

### PROTO_OPS

- `struct proto_ops* pops[] = {`
  - `PF_ATMPVC`
  - `atm_create`,
Connect time

Connect is called as follows:

```c
struct sockaddr_atmpvc addr;
addr.sap_family = AF_ATMPVC;
addr.sap_addr.itf = 0;
addr.sap_addr.vpi = 0;
addr.sap_addr.vci = 33;
rc = connect(s,(struct sockaddr *) &addr,
            sizeof(addr));
```

Internally within the generic connect system call

- The socket structure is recovered using the handle `s`
- The socket structure contains the proto_ops table pointer.
- The address of the routine `atm_do_connect` is found in that table
- `atm_do_connect` uses the interface number as an index in the `atm_dev` table where it finds pointers to driver data structures
- It copies the drivers device descriptor to the vcc structure
- It invokes the driver’s `ape25_open` function through the driver’s `dev_ops` table
After Connection

Generic Linux Networking
Linux ATM Networking
APE 25 Device Driver

ATM Device Table

struct atm_dev
  *vccs
  *atm_devops
  *dev_data

struct atm_devops =
  {
    ape25_open,
    ape25_close,
    :,
    ape25_send

struct pddtype ape
{
  pci data
  buffer pool data
  performance data
  atm_device_data

struct atm_vcc
  vci
  vpi
  *dev
  *dev_data
  *push
  *pop
  *sock
Device Driver Support for the Linux ATM Stack

Transmit buffer management

When an application calls............

\[\text{write}(\text{sock, buf, sizeof(buf)})\];

- `sys_write` determines this request is a socket write
- Socket structure contains a point to the ATM pvc `proto_ops`
- The `proto_ops` table points to `atm_sendmsg`

`atm_sendmsg`

- acquires skbuff(s) in kernel space and copies the message
- acquires the address of the driver send routine
**ape25_send**

- **Parameters**
  - `skbuff`
  - `vcc structure`
- Recovers pointer to “ape” structure from `vcc->dev_data`
- Invokes `atm_send5skb` in `atmxmit.c`

**atm_send5skb**

- Recovers a free TFD from the driver maintained free list
- Binds the skbuff to the TFD
- Stores the `vcc` address in an extension to the TFD
- Initiates the transmission.

**atm_xmitint**

- Removes TFD from the TCL as before
- Recovers `vcc` pointer from the TFD extension
- Calls the “pop” routine pointed to by the `vcc` to free the skbuff
Receive buffer management

- During initialization the driver allocates and binds skbuﬀs to the RBHs on the free list

```
APE 25
SRFL

RBH   RBH   RBH   RBH
SKBUFF SKBUFF SKBUFF

Driver
ape->erfl

RBH
SKBUFF
```
• During interrupt service the driver processes the Receive ready lists

Driver
ape->srrl[3]

APE 25
RRL_3

• RBH’s (4, 5, and 6) are returned to the free RBH list

• The LC value in the RBH is used to recover the VCC address from a table entry that is made at VCC open time.

• The address of the “push” routine in the VCC is used to forward the buffer to the protocol.
Free buffer list replenishment

- The Linux ATM protocol supports
  
  Driver managed receive skbuffs  
  Protocol managed receive skbuffs

- The APE 25 driver implements *driver managed skbuffs* and exports `ape25_free_rx_skb` in its `atm_devops` table

- The protocol calls `ape25_free_rx_skb` when the data in a receive buffer has been consumed.

`ape25_free_rx_skb`

- Dequeues a free RBH from the free rbh list
- Binds the RBH to the skb being freed
- Adds the RBH to the RFL for use by the APE 25.

Recovery from lost skbuffers

- The protocol doesn’t --always-- return the skbuffers.
- The driver keeps a count of *free* RBH’s.
- Free RBH count too high -> lost skbuffers.
- The driver replenishes the supply via `alloc_skb`
Reviewing the buffer lifecycle

1 - APE 25 consumes a RBH-SKBUFF via the SRFL register

2 - APE 25 produces on to RRL_n via the RRLn_LFDA register

3 - Driver (interrupt service routine) consumes from RRL_n via ape->srrl[n]. RBH is produced to free RBH list. Skbuff is “pushed” to protocol.

4 - Protocol returns skbuff to driver. Driver consumes RBH from free RBH list and produces matched pair to the free list via ape->erfl.