ATM Device Driver Development in Linux[•]

Alice L. Flower Robert M. Geist James M. Westall
aflower@cs.clemson.edu rmg@cs.clemson.edu westall@cs.clemson.edu

Department of Computer Science
Clemson University
Clemson, S.C. 29634-1906

1. Introduction

In this paper we describe our recent experiences in building a Linux device driver for an ATM NIC and integrating the device driver with version 0.31 of the Linux ATM protocol. We begin with an overview of ATM, AAL5, and the IBM Turboways ATM Network Adapter. In section 2 we present our development strategy and describe our experiences in building a standalone ATM device driver. In sections 3 and 4 we discuss the steps taken to provide support for the Linux ATM protocol and TCP/IP over ATM. We conclude with an assessment of the present state of the device driver and our plans for its future development.

1.1. ATM

Asynchronous Transfer Mode (ATM) is a promising, but still evolving, network technology designed to carry a heterogeneous traffic stream that includes voice, video, and data. An in-depth discussion of ATM is beyond the scope of this paper, and the interested reader is referred to textbooks on high speed networking such as [Stall:98]. ATM provides a connection-oriented service with best-effort delivery and can provide quality of service (QoS) guarantees with regard to both throughput and jitter control. An ATM connection between two network endpoints is called a Virtual Channel Connection (VCC). Two distinct connection types are supported. A Permanent Virtual Channel (PVC) is analogous to a leased line in the telephone network. It must be manually configured by a system administrator and persists until a system administrator manually destroys it. A Switched Virtual Channel (SVC) is analogous to a dialed connection in the telephone network. It is created in response to a request by one of the endpoints and persists only until one of the endpoints terminates it.

The 53 byte ATM cell is the basic unit of data transfer in an ATM network. To adapt the raw cell pipe provided by the ATM network to the specific requirements of various real-time and non-real-time applications, some standard transport-like protocols commonly called *ATM adaptation layers (AALs)* have been defined. These adaptation layers are present only in endpoint nodes of an ATM network. Five AALs have been specified to date. The most significant are AAL1 and AAL5.

AAL1 is designed to support real-time constant bit rate (CBR) traffic such as pulse code modulated (PCM) voice. AAL2 was initially intended for real-time variable bit rate (VBR) traffic. However, the final specification was so flawed as to render it unusable, and it has since been withdrawn. AAL3 and AAL4 were both designed for non-real-time VBR traffic (i.e., traditional data networks), and because the specifications were so similar in format and functionality, they were eventually merged creating AAL3/4.

[•] This work was supported by grant ATLCON-737 from the IBM Corporation.

AALs 1-4 were designed by the International Telecommunications Union (ITU), and their design reflected the ITU's historical focus on telephone (as opposed to computer) networks. The computer industry found AAL3/4 excessively complex and inefficient and proposed the Simple and Efficient Adaptation Layer (SEAL) as a replacement [Tanen:96]. SEAL was subsequently standardized as AAL5. Since AAL5 is now the de facto standard for all VBR traffic in ATM based computer networks, we do not consider the other AALs further in this paper.

1.2. Operation of AAL5

ATM adaptation layers are often viewed as consisting of two sublayers: the convergence sublayer (CS) and the segmentation and reassembly (SAR) sublayer. In AAL5 the function of the CS is to perform framing and error detection while SAR's job is to segment AAL5 frames into ATM cells for transmission and to reassemble AAL5 frames from ATM cells on reception.

We illustrate this process by an example. Suppose a 450 byte packet is passed by an upper layer protocol to the CS. The CS pads the packet (if necessary) so that its length is a multiple of 48, minus 8 (length $\equiv$ -8 mod 48). In this case 22 bytes of padding will be added yielding 472 bytes. The CS then adds an 8 byte trailer containing a 4 byte error detecting code and a 2 byte length field set in this case to 450. The padded packet with CS trailer appended is thus always an even multiple of 48 bytes in length and is sometimes called an AAL5 protocol data unit (PDU) or simply an AAL5 frame.

AAL5 frames are passed by the CS to the SAR sublayer for segmentation into 53 byte ATM cells. For AAL5 each ATM cell consists of a standard 5 byte cell header followed by a 48 byte payload. In this example, the SAR sublayer segments the 480 bytes of the AAL5 frame into 10 ATM cells, and then schedules them for transmission.

As cells arrive at the other end of the connection, the receiving SAR sublayer strips the 5 byte cell headers and catenates the 48 byte payloads in a reassembly buffer. When end of frame is indicated (by a bit in the last cell header), the reassembled frame is passed to the CS. The CS checks the frame for validity and passes the frame and a status indicator to the layer above it.

1.3. Support for ATM networks in Linux

Development of ATM protocol support in Linux is being coordinated by Werner Almesberger (the author of LILO) at the LRC (Laboratoire de Reseaux de Communication) in Lusanne, Switzerland. Although development is ongoing, ATM in Linux now provides most of the functionality expected in a state-of-the-art ATM network. Both permanent and switched connections are supported. ATM services are delivered to applications via a socket based API. The most rudimentary service is an unacknowledged datagram service that transports AAL5 frames over native ATM connections.

IP over ATM is also available and can be provided via two distinct mechanisms. Classical IP over ATM (CLIP) as defined in RFC 1577 [Laub:94] is supported via the atmarp daemon developed by Almesberger. LAN Emulation (LANE) as defined by the ATM Forum in af-lane-0021.000 [ATMF:95] was developed by Kiiskila Marko at the Tampere University of Technology in Tampere, Finland [Marko:96]. Either of these mechanisms

permits TCP/IP applications to operate transparently over an ATM infrastructure. Both CLIP and LANE use AAL5 as a transport.

The Linux ATM protocol support is completely independent of the characteristics of any underlying network adapter. To achieve this independence, the protocol depends upon the existence of a low-level, adapter-dependent device driver to implement a small set of functions (minimally: open connection; close connection; receive frame; and send frame).

1.4. The IBM Turboways 25 ATM PCI adapter

The IBM Turboways 25 ATM PCI adapter provides a low-cost but highly functional way to deliver ATM service to a user workstation. This adapter, based on the IBM APE25PCI (ATM Protocol Engine, 25 Mbps, Peripheral Component Interconnect) chipset, fully implements adaptation layers AAL1 and AAL5. Throughout the remainder of the paper we will refer to this NIC simply as the "APE25".

The APE25 supports eight distinct transmission queues in hardware. Each queue has an associated maximum sustained transmission rate, but short term high-rate bursts are also supported by a configurable token bucket admission control algorithm. When a VCC is created, it is bound to a specific transmission queue, and it may be permitted to transmit at the full rate of the queue or restricted to some fraction thereof.

Eight hardware receive queues are also supported. A device driver can choose to service them in a priority based order to give improved service to the VCCs bound to the higher priority queues.

2. Building a standalone ATM device driver

In this section we outline a procedure for building a standalone ATM device driver that supports an *ioctl()* based API using the standard Linux character device driver interface. Modern NICs, such as the APE25, fully implement their supported AALs in adapter microcode. Thus, the first challenge in building an ATM device driver lies in understanding the services that are provided and the interfaces that are exported by the AAL firmware. The second is building the software that interacts correctly and efficiently with those interfaces to deliver required services to the protocol layer that lies above the device driver. We often refer to this process as an exercise in *plumbing.*

We believe that it greatly expedites driver development to first build and *test thoroughly* a standalone driver *before* attempting to interoperate with the Linux ATM protocol. Building a standalone driver is a non-trivial undertaking in its own right. The IBM functional specification describing how a driver interacts with the APE25 is extremely well written but it is also over 200 pages long [IBM:96]. Thus, misunderstandings resulting in errors of omission and commission are inevitable.

The Linux ATM protocol is also quite large and complex. Even though its device driver interface is clean and well documented, it is still likely that errors will be made in interfacing the driver to the protocol. Because of the complexity of the protocol these errors can be quite difficult to track down and may require rebuilding the kernel with various debugging flags enabled. Simultaneously identifying and correcting errors in interactions with both the protocol and the NIC would be a daunting task indeed, and we recommend against it.

During the development of the standalone device driver we strongly recommend working in the PVC domain. Although, technically, there is nothing inherent in a device driver that associates it with either a PVC or SVC domain, operating in an SVC domain requires a functional signaling daemon. Thus, the same motivation for avoiding the Linux ATM protocol in the early stages of development also applies here. During the initial stages of development a *single system with PVC loopback* is the most productive environment!

2.1. Implementing Linux device drivers as loadable modules

Linux supports kernel extensions through dynamically loadable *modules*. This facility significantly reduces driver development time by reducing the number of kernel rebuilds and allowing installation and testing without system reboots. A module is a collection of C functions containing at least two functions with specific names, *init_module()* and *cleanup_module()*. The collection is compiled and linked as a single object file, say *atmdd.o*, and then loaded with the command */sbin/insmod atmdd.o*. The init_module() function is executed when the module is loaded, and the cleanup_module() function is executed when the module is removed, via the command */sbin/rmmod atmdd.o*.

In the initial stages of development, it may be necessary to perform a few kernel rebuilds. Only those kernel symbols explicitly exported for module use in */usr/src/linux/kernel/ksyms.c* are available, and it is common to find a need to export additional symbols. The names of these functions or structures must be added to the export list in ksyms.c, and a new kernel must be built to export them. When the symbol collection has stabilized, kernel rebuilding is no longer necessary.

Device drivers loaded as modules can often also be dynamically unloaded when errors are detected. This mechanism makes it possible to conduct the entire development cycle (test, identify error, repair, rebuild, and reinstall) without rebooting. Since device drivers normally do run with full kernel privileges, particularly severe errors can cause complete system crashes that do require rebooting. Nevertheless, we were amazed at the resilience of the kernel with respect to tolerance of fairly severe driver errors.

2.2. Defining the API

During its development, the standalone driver wears many hats. It serves as a learning tool, general testbed, and performance testing platform. Thus, the API that it exports should not necessarily reflect what one would expect to see in a "production" device driver. The first four functions exported by our driver are just what one might expect: open; close, read, and write.

- AIO_CREATEVCC Create a PVC based VCC
- AIO_FREEVCC Close a PVC based VCC
- AIO_RECVDATA Receive an AAL5 frame
- AIO_SENDDATA Send an AAL5 frame

The next five functions essentially export total control of the APE25 to the application level. One would clearly not want to do this in a production device driver (without at least building a root access filter into the device driver.) However, in a development environment it greatly facilitates the construction of useful application level diagnostic tools.

- AIO_RDCREG Read one of the APE25's control registers
- AIO_WTCREG Write one of the APE25's control registers
- AIO_RDSAP Read one of the APE25's system access port registers
- AIO_WTSAP Write one of the APE25's system access port registers
- AIO_RDPMD Read device driver capture performance measurement data

For example our *apestat* utility dumps the contents of all the APE25's control registers:

```
MODE_REG             0100   - 1741
ACONFIG_REG          0102   - ffff
ERROR_REG            0106   - 1000
TCLP0_RCELL          0108   - 0069
TCLP1_RCELL          010a   - 0000
THB_ER_CELL          010c   - 0011
THB_ER_DC            010e   - 0011
MISROUTE_CELL        0110   - 7e90
TCLP0_TCELL          0118   - 0000
TCLP1_TCELL          011a   - 02ab
   :                   :        :
```

In like manner the *apelc* utility formats and dumps the 128 byte control block used by the APE25 to represent the state of a connection and *apepmd* formats and dumps the performance measurement data captured by the device driver.

2.3. Standalone ATM driver initialization

Since the standalone device driver exports an ioctl() based API, it is necessary to create a Linux character device through which it may be accessed. The *mknod* command is used to create a device node with any previously unused name and major device number:

```
mknod /dev/ape25 c 15 1
```

The first step in the *init_module()* function of the standalone driver that supports an ioctl() API is to register the driver as a Linux character device. Registration is done by passing the following data structure to register_chrdev().

```
static struct file_operations atm_fops =
{
   NULL,          /* lseek              */
   NULL,          /* read               */
   NULL,          /* write              */
   NULL,          /* readdir            */
   NULL,          /* select             */
   atm_ioctl,     /* ioctl              */
   NULL,          /* mmap               */
   atm_open,      /* open               */
   atm_release,   /* release(close)     */
   NULL,          /* fsync              */
   NULL,          /* fasync             */
   NULL,          /* check_media_change */
   NULL           /* revalidate         */
};
```

```
int init_module(void)
{
      :
   printk("IBM APE25: Module initialization\n");
   if (rc = register_chrdev(15, "ape25", &atm_fops))
   {
      printk( "APE25: cannot register; received %d\n", rc);
      return 1;
   }
```

The first two operands of `register_chrdev()` correspond to the values specified in the *mknod* command. The last is a pointer to a table of function addresses. As can be seen from the table above, our standalone driver exports only open, close, and ioctl entry points.

The next step in driver initialization is to attempt to determine which PCI device is the ATM adapter. This is done by serially checking all PCI devices until one carrying the manufacturer's standard vendor and product id is found.

```
   for(pci_dev = pci_devices; pci_dev; pci_dev = pci_dev->next)
   {

      if (pci_dev->vendor==PCI_VENDOR_ID_IBM &&
          pci_dev->device==PCI_DEVICE_ID_ATM_APE25)
      {

      /* Save relevant data */

         ape->bus    = pci_dev->bus->number;
         ape->devfn  = pci_dev->devfn;
         found = 1;
         break;
      }
   }
```

When the device identity has been established, Linux PCI utility routines can be used to read necessary configuration data including the I/O base address, local memory base address, and interrupt number being used by the device.

```
   rc  = pcibios_read_config_dword(bus, devfn, PCI_BASE_ADDRESS_0,
                                   &ape->pci_iobase);
   rc |= pcibios_read_config_dword(bus, devfn, PCI_BASE_ADDRESS_1,
                                   (int *)&ape->pci_membase);
   rc |= pcibios_read_config_byte(bus, devfn, PCI_INTERRUPT_LINE,
                                    &ape->pci_irq);
```

To direct device interrupts to the interrupt service routine included in the driver module the *request_irq()* function is invoked.

```
   rc = request_irq(ape->pci_irq, ape25_irq, SA_INTERRUPT, "ape25", ape);
```

Up to this point the initialization code is largely NIC independent, but now device dependent initialization must be done. For the APE25 three entities must be initialized: connection management structures; transmit data structures; and receive data structures.

2.3.1. Initialization of connection management structures

The APE-25 contains a local data memory of at least 16 KBytes in size. This memory, called XRAM, contains the data structures through which the device driver exchanges information with the APE-25's microcode. The layout of the data structures contained in XRAM is completely unconstrained and must be defined by the device driver designer. The locations of the individual data structures are conveyed to the microcode via control registers that must be set up during driver initialization. Our present XRAM layout is shown in table 1.

Element Type	Offset in XRAM (hex)	Element size (bytes)	Number of Elements
VP_TABLE_BASE	0000	2	256
LCI_TABLE_BASE	0200	2	64
TCB_TABLE_BASE	0280	62	2
R1CB_TABLE_BASE	0300	64	16
R5CB_TABLE_BASE	0700	64	72
ROCB_TABLE_BASE	1900	64	15
T1RQ_TABLE_BASE	1cc0	8	16
T5RQ_TABLE_BASE	1d40	8	24
LC_TABLE_BASE	1e00	64	128

Table 1: XRAM layout.

IBM uses the term Logical Channel (LC) to refer to a (potential) VCC, and each LC must be represented by a 128 byte data structure in XRAM. Therefore, an adapter with 16KB of XRAM has a hard upper bound of 128 (16K / 128) on the number of open VCCs. Since XRAM is also used for cell buffers and other transmit and receive data structures, the actual upper bound is a good bit smaller in practice.

A flexible, two-level table lookup mechanism is used by the microcode to map the (virtual path identifier (vpi), virtual circuit identifier(vci)) numbers carried by arriving cells to LC addresses. The mapping tables, called the Virtual Path (VP) table and the Logical Channel Index (LCI) table must be constructed by the device driver, and they implicitly determine which (vpi, vci) address ranges are actually usable. Our driver is presently configured to support only vpi 0, and the vci range is 0 - 63. Later in the paper we describe an interesting problem arising from these constraints on address range.

2.3.2. Initialization of transmit data structures

Since ATM NICs interact with system software at a relatively high level, it is necessary that buffer descriptors have a standard format that is understood by both the device driver and the microcode. A standard buffer header called the Transmit Frame Descriptor (TFD) is defined by the APE25 and *must* be used by the device driver in passing frames to the microcode. This buffer header is somewhat complex because the APE25 supports multiple mechanisms for initiating a transmission. Each TFD represents a single AAL5 frame. Hardware scatter/gather support is fully implemented by the APE25, and so each TFD also contains a variable length array of

pointer/length fields that identify the fragments of the frame. Finally, the TFDs themselves may be chained together so that the transmission of multiple frames can be initiated via a single start request to the APE25. Our driver implements neither scatter/gather nor frame chaining at the present time.

During initialization the device driver uses kmalloc() to allocate a pool of TFDs and links them together on a driver-managed free TFD list (FTFDL). The standalone driver also allocates a pool of transmit buffers and permanently binds one of them to each TFD.

The device driver must next create two XRAM resident tables that are used in conveying transmission requests to the APE25. These tables are called Transmit Ready Queues (TRQs) and they are managed as circular buffers with the device driver acting as the producer and the APE25 as the consumer. TRQ1 is used for AAL1 and TRQ5 for AAL5.

When the device driver wishes to initiate a transmission, it writes the LC number and the address of the Transmit Frame descriptor (TFD) into the next available TRQ element. Thus, each TRQ table element represents a pending transmission request. Since it is imminently possible to issue transmit requests faster than the APE25 can transmit the associated frames, the APE25 contains a status register with bits that indicate when either TRQ is full. How the driver should respond to a send request when the TRQ is full, poses an interesting problem that will also be addressed later. We presently allocate 24 TRQ elements for TRQ5.

When the APE25 completes the transmission of an AAL5 frame, it needs a mechanism for returning control of the TFD and associated frame buffer to the device driver. The Transmit Complete List (TCL) serves this purpose. The second word in the TFD serves as the TCL link field. When the APE25 completes the segmentation and transmission of an AAL5 frame, it appends the TFD to the TCL and generates a transmission complete interrupt. The APE25 accesses the end of the TCL through the TCL_LFDA (last frame descriptor address) control register. To initialize the TCL the driver removes a TFD from the free TFD list, sets its TCL link field to null, and loads the TCL_LFDA register with the address of that TFD.

The Transmit Cell Buffer (TCB) table is an array of 64 byte buffers that resides in XRAM and is used by the segmentation microcode to construct ATM cells. The single TCB array services AAL1, AAL5, and Operations, Administration and Maintenance (OAM) queues. Since we support only AAL5 and were unable to see any motivation for doing otherwise, we allocate only two transmit cell buffers.

2.3.3. Initialization of receive data structures

A standard buffer header called the Receive Buffer Header (RBH) is also defined by the APE25. As with the TFDs, the device driver uses kmalloc() to allocate a pool of RBHs and the standalone driver also allocates a pool of AAL5 frame buffers and binds one to each RBH. Several RBH lists through which the driver and the APE25 communicate must then be initialized.

The APE25 supports 8 hardware receive queues known as Receive Ready List 0 (RRL0) through RRL7. Each LC is bound to one of these queues via a field in its 128 byte LC descriptor. When reassembly of an AAL5 frame completes, the APE25 appends the RBH to the appropriate RRL and generates an interrupt. The device driver can identify which RRLs need service via a bitmap in the interrupt status register. To initialize RRL *n* the device

driver takes a free RBH, sets its link field to null and loads the RRLn_LFDA (last frame descriptor address) register with the address of that RBA.

The device driver must also construct a list of free RBHs that are bound to AAL5 frame buffers. This list is called the Receive Free List (RFL) and the APE25 consumes buffers from it as frames are reassembled. The APE25 supports a scatter/gather mode of operation in which elements of a single large AAL5 frame may be reassembled into multiple RBHs and frame buffers, but the device driver does not now support this mode.

Receive cell buffer arrays must also be created in XRAM by the device driver. Each cell buffer is 64 bytes in size, and the APE25 uses three dedicated cell buffer arrays for the reception of AAL1, AAL5 and OAM cells respectively. When the cell buffer array becomes full, or the RFL becomes empty, the APE25 will generate an interrupt to notify the driver of the specific condition and will simply drop frames until the condition is corrected. We now use an allocation of 72 receive cell buffers and 48 RBHs on the RFL. Under these initial conditions, the standalone device driver almost never incurs frame loss due to cell buffer or RFL depletion.

2.3.4. Activating the APE25

The next steps in the initialization procedure are to load the microcode used by the APE25's transmit and receive engines and to initialize some miscellaneous control registers. The APE25 is th finally activated through the sequence shown here.

```
/* Finally init the interrupt mask reg, the transmit enable */
/* (ACONFIG) reg and turn on the board with the mode reg.   */

   atm_wtsreg(ape, MISR, 0xffff);
   WT_CNTLREG(ape, ACONFIG_REG, 0xffff);
   WT_CNTLREG(ape, MODE_REG,  NORMAL_MODE | PICO_ENABLE);
```

2.4. Transmitting and receiving packets with the standalone driver

A useful way to view the operation of any ATM device driver is as a parallel collection of traditional producer/consumer procedures. The standard "problems" in any such environment are synchronization and mutual exclusion. The synchronization mechanism must ensure that a consumer is blocked when there is no item to be consumed and that a producer is blocked when there is no available buffer space. The mutual exclusion mechanism must ensure that producers and consumers don't interact with shared data structures in a destructive way. We examine the operation of both transmit and receive from this perspective.

2.4.1. Transmitting AAL5 frames

Transmission involves three cooperating processes: the device driver's send frame routine; the transmit engine running on the APE25; and the device driver's transmit interrupt service routine. All three elements function as both consumers and producers. When the device driver's send frame routine receives a request to transmit a packet, it must consume a TFD and its associated frame buffer from the Free TFD List. It accesses the TFD via the pointer `ape->sftfdl`, removes it from the head of the list, copies the user's data to the frame buffer, and then produces a transmission request into the TRQ5 table. The transmit engine of the APE25 consumes TRQs from the TRQ5 table, segments and transmits the associated frames, and then produces free TFDs onto the tail of

the TCL. The device driver's interrupt service routine consumes free TFDs from the head of the TCL via the pointer `ape->stcl` and produces them onto the tail of the Free TFD List using the pointer `ape->eftfdl`. The complete buffer lifecycle is shown in figure 1. A link value of 1 is defined by the APE25 to denote the end of a buffer header list.

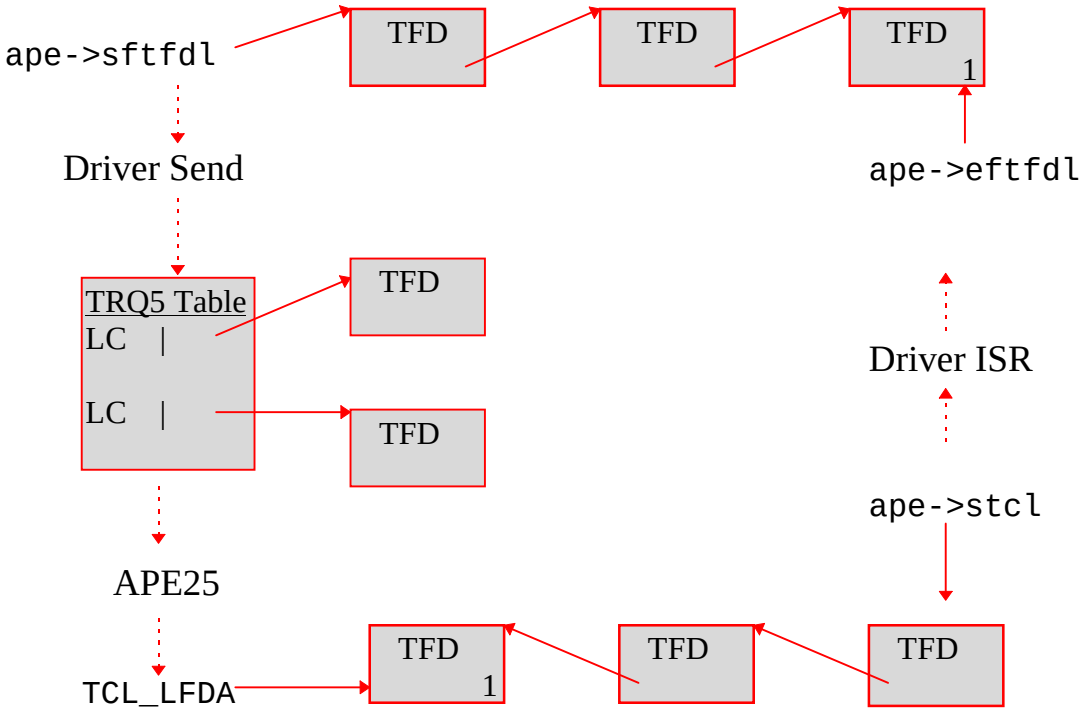

Figure 1: The transmit buffer lifecycle.

2.4.1.1. Mutual exclusion

An interesting mechanism for avoiding mutual exclusion problems in buffer list management is implemented by the APE25. The device driver also uses the mechanism in managing lists such as the Free TFD List that do not directly involve the APE25. The following rules govern all operations on buffer header lists:

- Consume from the head of the list.
- Produce onto the tail of the list.
- Never consume the only remaining element in the list.

These rules ensure that:

- Only the producer alters a tail pointer such as `ape->eftfdl.`
- Only the producer alters the link pointer of a list element.
- Only the consumer alters a head pointer such as `ape->sftfdl.`

Since each list has only a single producer and consumer, these rules ensure that there are no shared data elements upon which the producer and consumer might destructively interact, and no additional mutex mechanisms are required or used.

2.4.1.2. Synchronization

A synchronization problem does arise in the device driver send routine. If the link field of the first element of the Free TFD List indicates that the list contains only a single element, the driver must not consume the element. Also, if the transmit status register of the APE25 indicates that the TRQ5 table is full, then the driver cannot attempt to initiate the transmission.

In the absence of some catastrophic failure, both of these situations are transient and will resolve themselves as soon as some pending transmissions complete. Thus the standalone device driver has several possible courses of action:

- Sleep until awakened by the transmit complete interrupt handler.
- Busy wait until the condition is resolved.
- Return without transmitting the packet.
- Implement some sort of transmit queueing mechanism.

In a standalone device driver the best strategy is to sleep. However, as we shall see, when TCP/IP over ATM is in use sleeping is not an option, and all of the other options have some apparent disadvantages as well.

2.4.2. Receiving AAL5 frames

As was the case with transmission of frames, reception also involves three cooperating processes: the receive engine running on the APE25; the device driver's receive interrupt service routine; and the device driver's receive frame routine. All three processes function as both consumers and producers.

The buffer lifecycle also includes three buffer lists. As cells arrive, reassembly buffers are consumed by the APE25 from the Receive Free List (RFL) as shown in figure 2.

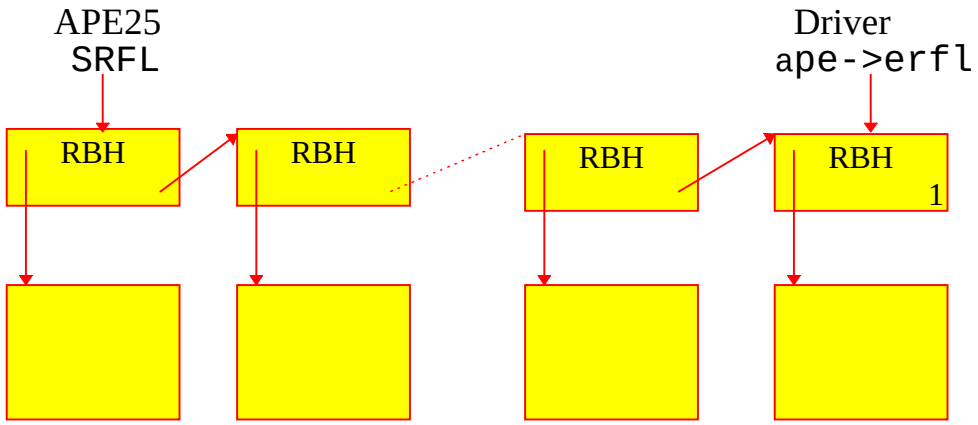

Figure 2: The Receive Free List.

When a frame has been assembled, the APE25 appends it to the end of the Receive Ready List (RRL) associated with the LC to which the frame was sent. Figure 3 depicts the state of RRL_3 after three frames have been assembled by the APE25 but not yet consumed by the device driver.

Since there are only 8 RRLs, it may be necessary to assign multiple LCs to a common RRL. Therefore, the driver maintains an array of 63 RBH lists called the LC Receive Lists. One list is dedicated to each possible LC. When the standalone device driver services the receive ready interrupt, it must perform a demultiplexing function, consuming the elements from the RRL and producing them onto the list associated with the destination LC.

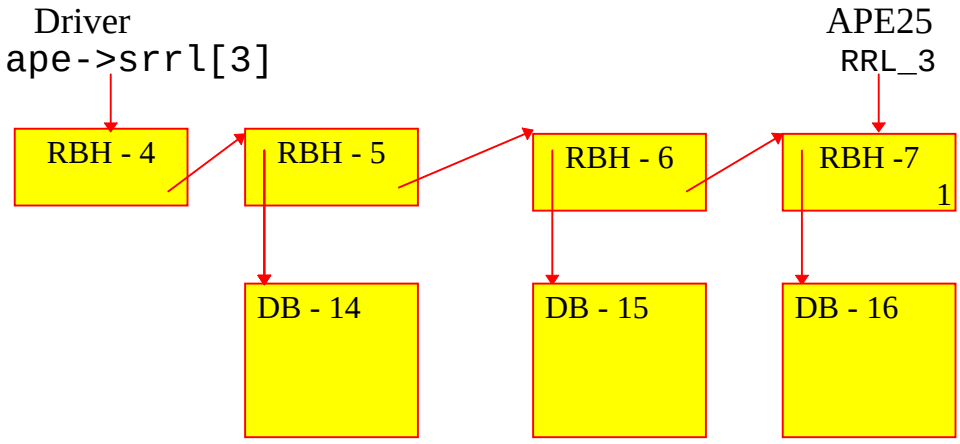

Figure 3: The Receive Ready List (RRL).

One factor slightly complicates this procedure. In the situation shown in figure 3, the driver needs to consume the frame buffer identified as DB-16, but it cannot consume RBH-7 to which DB-16 is bound without violating the mutual exclusion mechanism. However, RBH-4, the original start/end of list placeholder is not bound to any frame buffer.

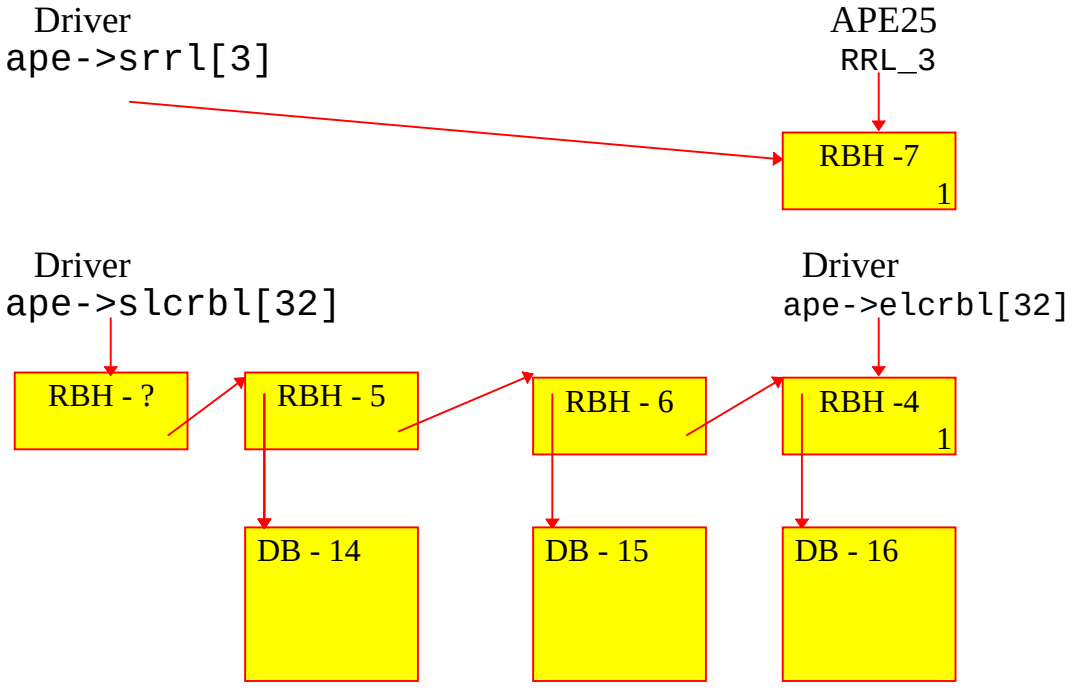

Figure 4: Migration of buffers to an LC receive list.

Therefore, when the driver services any RRL it must always rebind the last frame buffer to the placeholder RBH that is always found at the head of the list. The last RBH in the RRL becomes the new placeholder after the rebinding. Assuming that all the buffers shown in figure 3 were destined for LC 32, this process is illustrated in figure 4.

Buffers are consumed from the per LC receive lists by the device driver's receive frame routine. If an application makes a receive request for a particular LC and the associated LC receive list contains only the single placeholder RBH, then the driver sleeps on a wait queue associated with the LC. When driver's receive interrupt service routine enqueues a frame buffer on any LC's receive list, it also issues a wakeup for the LC's wait queue.

The receive frame routine also uses the rebinding technique that was illustrated in figure 4 in consuming RBHs and frame buffers from the LC receive list. When the data has been copied from the frame buffer to the user provided buffer, the RBH and the frame buffer are appended to the tail of the RFL. The entire receive buffer lifecycle is shown in figure 5.

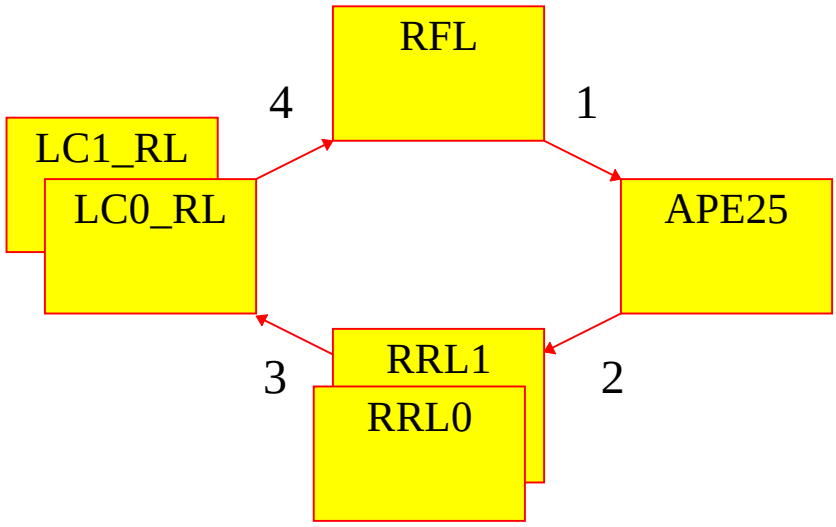

Figure 5: The receive buffer lifecycle.

1 - The APE25 consumes a buffer via the `SRFL` register.

2 - The APE25 produces onto `RRL_n` via the `RRLn_LFDA` register.

3 - The interrupt service routine consumes from `RRL_n` via `ape->srrl[n]` and produces to LC_RBL_k via `ape->elcrbl[k].`

4 - The device driver receive routine consumes from LC_RBL_k via `ape->slcrbl[k]` and produces to the free list via `ape->erfl.`

3. Interfacing the standalone device driver to the Linux ATM protocol

The interface between Almesberger's Linux ATM protocol and NIC dependent device drivers is simple, clean, and well documented [Almes:96a]. It should take no more than a few days to connect a functional standalone driver to the protocol. The changes are so minimal that we recommend encapsulating them in `#ifdef`'s so that one code base can be used to build both standalone and integrated drivers.

In this section we consider interacting only with Almesberger's socket based native ATM service, which provides a best-effort transport of AAL5 frames over native ATM connections [Almes96b]. Support for TCP/IP over ATM is discussed in section 4. Throughout this section we use the term *protocol* to refer to the native ATM service of Almesberger's Linux ATM protocol.

Changes required in the initialization of the device driver are described in sections 3.1 and 3.2. In the remainder of section 3 we describe how the device driver interoperates with the protocol in opening VCCs, sending frames, and receiving frames.

3.1. Initialization of buffer management structures

Some modifications were required in the initialization of transmit and receive buffer management structures. When the protocol initiates a transmission, it passes a standard Linux socket buffer (*skbuff)* structure to the driver. The application data has already been copied to this kernel space data structure at the time the driver is called. Therefore, the data buffers allocated and bound to the TFDs by the standalone driver are no longer needed, and the associated code was eliminated via the `#ifdef` mechanism.

An analogous situtation exists in the initialization of the receive free list (RFL). The standalone driver allocates the receive frame buffers that populate this list by directly calling kmalloc(). Since it is necessary to pass Linux skbuffs to the protocol, the kmalloc() was replaced by alloc_skb(). The updated buffer allocation and binding code is shown below. Since our device driver does not presently support the APE25's receive chaining capability, the maximum size frame that can be received is RB_SIZE - 8. (The APE25 does place the CS trailer in the buffer.)

```
for (i = 1;  i < SKB_COUNT; i++)
{

/* Get a Free RBH from the driver list and then ask the */
/* system for an SKB.                                   */

   nrbh = atm_getrbh(ape);
   skb = alloc_skb(RB_SIZE, GFP_KERNEL);
   if (skb == 0) return(-1);

/* Hook the RBH to the SKB and the RBH to the RFL */

   skb->len   = RB_SIZE;
   nrbh->data = (char *)skb->data;
   nrbh->len  = RB_SIZE;
   nrbh->skb  = skb;
   rbh->next  = nrbh;    /* rbh tracks end of RFL  */
   rbh = nrbh;           /* so update it here.     */
}
```

3.2. Binding the device driver to the protocol

The last action of the init_module() function is to bind the device driver to the protocol. The device driver's ape25_register() function initiates the binding by passing a table of 14 function pointers to the protocol's atm_dev_register() function.

```
static struct atmdev_ops atm_ops =
{
  ape25_open,           /* open         */
  ape25_close,          /* close        */
  ape25_ioctl,          /* ioctl        */
  ape25_getsockopt,     /* getsockopt   */
  ape25_setsockopt,     /* setsockopt   */
  ape25_send,           /* send         */
  NULL,                 /* sg_send      */
  NULL,                 /* poll         */
  NULL,                 /* send_oam     */
  NULL,                 /* phy_put      */
  NULL,                 /* phy_get      */
  NULL,                 /* feedback     */
  NULL,                 /* change_qos   */
  ape25_free_rx_skb,    /* free_rx_skb  */
};

/**/
/* Attempt to register the APE25 device */

int ape25_register(
struct pddtype *ape)    /* Pointer to device dependent descriptor */
{
   int i;

   ape->atmdev = atm_dev_register("ape25", &atm_ops, 0);
   if (ape->atmdev == NULL)
   {
       printk("ape25: Can't register device \n");
       return -EIO;
   }

   printk("ape25_register: Free callback at %x \n", atm_ops.free_rx_skb);

/* Get the ESI from the adapters nvram */

   printk("ape25_register: ESI is: ");
   for (i = 0; i < ESI_LEN; i++)
   {
      ape->atmdev->esi[i] = atm_rdnvram(ape, ESI_OFFSET + i);
      printk("%02x.", ape->atmdev->esi[i]);
   }
   printk("\n");

   ape->atmdev->ci_range.vpi_bits = 0;
   ape->atmdev->ci_range.vci_bits = LC_BITS;

/* The 'ape' data structure is our key to the world... We must be able */
/* to recover it from the atm_dev structure in subsequent open, close  */
/* and send calls.                                                     */

   ape->atmdev->dev_data = ape;
   return(0);
}
```

Most of these function pointers can be null. At present ape25_ioctl, ape25_getsockopt, and ape25_setsockopt are essentially null functions and, in retrospect, ape25_free_rx_skb probably should have been.

If registration succeeds a pointer to a structure of type atm_dev is returned to the device driver. We fill in the end system address, number of valid vci bits, and most importantly a pointer to our own root data structure which is pointed to by the variable *ape*.

At this point the initialization of the driver is complete and transfer of AAL5 frames should be possible. In the remainder of this section we provide an outline of how the device driver interoperates with the protocol in the initialization of a VCC and in the sending and receiving of AAL5 frames.

3.3. VCC initialization

The protocol supports the Berkeley sockets API. An ATM VCC is created when an application calls the *connect()* function. The device driver is then invoked by the protocol at its open entry point *ape25_open()*. Most of the code shown here is prescribed in Almesberger's Linux ATM Device Driver Interface document [Almes96a].

```
int ape25_open(
struct atm_vcc *vcc,    /* vcc structure from driver    */
short vpi,              /* vpi... better be 0!          */
int vci)                /* vci... better be < LC_COUNT  */
{
   struct pddtype *ape;
   int             rc;
   int             lcnum;
   int             lcndx;

/* Recover pointer to our root data structure */

   ape = vcc->dev->dev_data;

/* Indicate that we don't support single copy receives (SCRX) */

   vcc->flags &= ~ATM_VF_SCRX;
```

Here we resolve the (vpi, vci) to be used in the connection. The function `atm_find_ci()` is supplied by the protocol and will attempt to return a usable (vpi, vci) pair when the application has left the values unspecified. It is mandatory that the driver set the `ATM_VF_ADDR` flag bit as soon as the (vpi, vci) address is considered valid.

```
   rc = atm_find_ci(vcc, &vpi, &vci);
   if (rc)
      return(rc);

   vcc->vpi = vpi;
   vcc->vci = vci;

   if (vci != ATM_VPI_UNSPEC && vpi != ATM_VCI_UNSPEC)
      vcc->flags |= ATM_VF_ADDR;

   if (vcc->aal != ATM_AAL5)
      return(-EINVAL);
```

The remainder of the initialization involves setting up the internal data structures of the device driver that represent the VCC. The function atm_initlc5() performs the initialization of the associated 128 byte LC structure in the APE25's XRAM. The table, vcctab[], is later used by the device driver to identify the vcc structure to which each incoming frame belongs. The APE25 converts the (vpi, vci) of an arriving frame to an LC index and stores it in the RBH. The device driver can then recover the vcc pointer by a single lookup in the vcctab.

```
   rc = atm_freelc5(ape, vpi, vci);
   lcnum = atm_initlc5(ape, vpi, vci);

/* If lc was obtained... bind vcc to lc in vcc table */

   if (lcnum > 0)
   {
     lcndx = lcnum - FIRST_LCID;
     vcctab[lcndx] = vcc;
     vcc->vci = vci;
     vcc->vpi = vpi;
     vcc->flags |= ATM_VF_READY;
   }
   else
   {
      vcc->flags &= ~ATM_VF_ADDR;
   }

   if (lcnum > 0)        /* initlc returns lcnum on success */
      return(0);
   else
      return(lcnum);
}
```

3.3 Frame transmission

The protocol calls the device driver at its send entry point for each AAL5 frame to be sent. The interface is as shown:

```
int ape25_send(
struct atm_vcc *vcc,     /* -> Protocol's VCC structure */
struct sk_buff *skb)     /* Buffer to be sent.          */
```

Before the data can be sent the driver must consume a free TFD and ensure that there is space available in the transmit ready queue. The designer is again confronted with the sleep, busy-wait, or return dilemma that was presented in the context of the standalone driver. When IP over ATM is *not* being used, then sleeping works fine and remains the best option. Thus, the complete transmit synchronization code now looks like:

```
/* We cannot initiate our transmission until we    */
/* find a free TFD and the TRQ is not full         */

   tfd = ape->sftfdl;
   while ((tfd->tclnext == (struct tfdtype *)1) ||
          (atm_rdsreg(ape, AAL_STATUS) & 1) )   ||
          (ape->tbcount[lcndx] >= MAX_XBCOUNT))
   {
      interruptible_sleep_on(&ape->tfd_waitq);
      tfd = ape->sftfdl;
   }
```

The third test, involving ape->tbcount[lcndx], is necessary to enforce QoS policies when there are competing processes whose VCCs have significantly differing sustainable throughput guarantees. The test limits the number of TFDs that can be held by a process any one time. In the absence of this limit, a process that has a low sustained throughput guarantee but tries to send at a very high rate will consume all available TFDs. When this situation occurs, *all* processes, including those with high sustained rates, will compete on an equal basis for a newly freed TFD. As a result the effective throughput of all processes is limited to that of the process with the low throughput guarantee.

The remainder of the transmission sequence is straightforward. A TFD is consumed from the free list and initialized, the skbuff is bound to it, and the request is placed on the TRQ.

```
   ape->sftfdl = tfd->tclnext;       /* Consume free TFD */

   atomic_inc(&ape->tbcount[lcndx]);
   tfd->next     = (struct tfdtype *)1;
   tfd->tclnext  = (struct tfdtype *)1;
   tfd->lci      = lcid;
   tfd->prmstat  = 0;
   tfd->bufcount = 1;
   tfd->control  = 0;

/* Set up length and buffer pointer.. Not using scatter */
/* gather so sdulen = buf[0].len..                       */

   tfd->sdulen         = skb->len;
   tfd->dbds[0].buflen = skb->len;
   tfd->dbds[0].data   = skb->data;
   tfd->skb            = skb;
   tfd->vcc            = vcc;

/* Finally pass the request to the APE25 */

   loc = (unsigned long)tfd;
   atm_wtsreg(ape, TRQ_AAL5_LC, lcid);
   atm_wtsreg(ape, TRQ_AAL5_TFDA_HI, (loc >> 16) & 0xffff);
   atm_wtsreg(ape, TRQ_AAL5_TFDA_LO, loc & 0xffff);
   return(0);
```

When the transmission has completed, the APE25 places the TFD onto the TCL and generates an interrupt. The interrupt service routine must return the TFD to the free list and the skbuff to the protocol. The recommended mechanism for returning an skbuff is to invoke the pop() routine provided by the protocol. Therefore, the APE25-defined TFD structure was augmented to hold pointers to the vcc and skbuff structures. The code used to consume TFDs from the free list and return the associated buffers to the protocol is shown below.

```
   do
   {
      ltfd  = tfd;
      tfd   = tfd->tclnext;
      lcndx = tfd->lci - FIRST_LCID;
      atomic_dec(&ape->tbcount[lcndx]);
```

```
   /* If this descriptor seems to have an attached skbuff */
   /* then free the skbuff.                               */

      if (tfd->skb != 0)
      {
         if (tfd->vcc->pop == 0)
            dev_kfree_skb(tfd->skb, FREE_WRITE);
         else
            tfd->vcc->pop(tfd->vcc, tfd->skb);
      }
   }
   while (tfd->tclnext != (struct tfdtype *)1);

/* Finally copy the consumed descriptors to the free list */

   ltfd->tclnext = (struct tfdtype *)1; /* Set end of free list     */
   ape->eftfdl->tclnext = ape->stcl;    /* Attach to free list      */
   ape->stcl = tfd;                     /* Update TCL   list start  */
   ape->eftfdl =  ltfd;                 /* Update free  list end    */

/* Finally wake up processes sleeping in resource waits */

   wake_up_interruptible(&ape->tfd_waitq);
}
```

3.4 Frame reception

As described in section 3.1, during initialization the device driver enqueues a collection of RBH structures with bound skbuffs to the RFL. When the first cell of a new frame arrives, the APE25 consumes a buffer from the RFL and reassembles the frame in that buffer. When the reassembly is complete, the APE25 enqueues the RBH onto the RRL to which the ATM VCC of the arriving frame was bound and generates a receive complete interrupt. A simplified view of the functions performed by the interrupt handler is shown in the code fragment below.

```
   rbh = ape->srrl[qid];  /* Recover ptr to target RRL */
   while (rbh->next != (struct rbhtype *)1)
   {
      nrbh = rbh->next;
      lcid = rbh->lci;
      len  = rbh->sdulen;
      lcqid = lcid - FIRST_LCID;
      vcc = vcctab[lcqid];

      if (vcc != 0)
      {
         if (rbh->skb)
         {
            rbh->skb->atm.vcc = vcc;
            rbh->skb->len = len;
            rbh->skb->atm.size = len + 8;   /* Include CS trailer */
            vcc->push(vcc, rbh->skb);
         }
         rbh->skb = 0;
      }
      atm_freerbh(ape, rbh);
   }
```

The device driver converts the rbh->lci field provided by the APE25 to an index in the device driver's vcctab[] where a pointer to the protocol's vcc structure was saved at the time the ATM VCC was opened. After fields in

the skbuff header are filled in, the skbuff is forwarded to the protocol layer above by calling the push function pointer in the vcc structure. Finally, the free RBH is returned to the free RBH list.

Since buffers are continually consumed from the RFL by the APE25, a mechanism for replenishing the RFL is clearly required. In the following we discuss some possible alternatives. The standalone device driver allocates its pool of frame buffers at initialization time and internally manages them thereafter. The protocol also permits the device driver to manage the receive buffer pools by providing a mechanism for returning skbuffs to the device driver after the associated data has been consumed. To use this mechanism a device driver must simply include a free_rx_skb() function in the atmdev_ops table passed to the protocol when the device driver registers itself. We chose to use this approach on the theory that a device driver can manage a list of free buffers much more efficiently than is possible using alloc_skb() and dev_kfree_skb() because of their reliance on kmalloc() and kfree(). In retrospect, because of factors that will be described, the use of device driver managed receive buffers was probably a bad choice.

The device driver's free_rx_skb function is ape25_free_rx_skb(). When passed a free skbuff by the protocol, it dequeues an RBH from the Free RBH list, binds the RBH to the skbuff being returned, and enqueues the RBH on the RFL. This operation, illustrated in the code below appears to "close the loop" as far as receive buffer management goes.

```
/* Put a newly free skb back on the receive free list */

void ape25_free_rx_skb(
struct atm_vcc *vcc,
struct sk_buff *skb)
{
   struct pddtype *ape;
   struct rbhtype *rbh;

   ape = vcc->dev->dev_data;  /* Recover ptr to driver data*/

/* Allocate a free buffer header to associate with */
/* the sk buff.                                    */

   rbh = (struct rbhtype *)atm_getrbh(ape);

/* Rebind the buffer to the skb */

   skb->len  = RB_SIZE;
   rbh->data = (char *)skb->data;
   rbh->len  = RB_SIZE;
   rbh->skb  = skb;

/* Append it to the RFL */

   ape->erfl->next = rbh;
   ape->erfl = rbh;
}
```

In our initial testing, this buffer management strategy appeared to function correctly with the protocol. Stress testing subsequently revealed an occasional loss of skbuffs. That is, buffers were "pushed" to the protocol, but were never returned back to the device driver! The behavior appears to occur only when a receiving process is terminated by a signal before consuming all enqueued skbuffs. As the steady state length of the RFL is depleted, the rate at which frames are dropped under heavy loads increases. We received an even greater surprise when

testing IP over ATM. We found that skbuffs are *never* "popped" back to the device driver! The second complicating factor is that the rate at which buffers are returned to the driver is limited to the rate at which data are consumed by the application. Under heavy system loads and high incoming frame rates we found that it was easily possible to totally deplete the RFL. When this situation occurs, the APE25 will simply drop frames until the RFL is replenished.

Therefore, it was necessary to modify the original closed loop designe of the device driver and incorporate a demand-based mechanism for dynamically allocating and freeing skbuffs. This mechanism works by maintaining a count of the number of elements on the Free RBH list. Since the RBH structures are totally under the control of the device driver, no leakage occurs. Since RBH structures reside only on the Free RBH list or on the RFL, the number of elements on the Free RBH list also determines the number of elements on the RFL. Each time a receive operation completes, the device driver computes the number of elements on the RFL. If the length of the RFL reaches a low-water-mark equal to one-half of its length at initialization, the device driver allocates new skbuffs until the RFL returns to its original length.

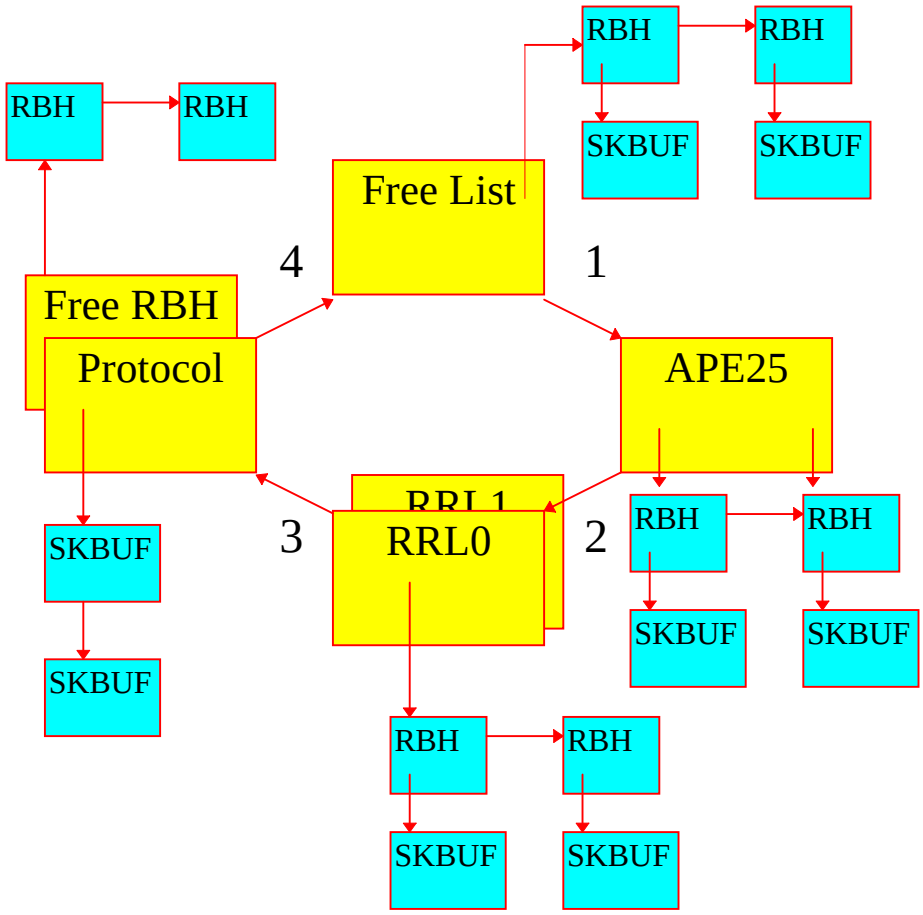

Figure 6: The receive buffer lifecycle.

In the case of severe, short term internal congestion it is possible to end up with all available memory dedicated to skbuffs. As the congestion abates, the skbuffs are gradually returned to the device driver's free_rx_skb() function by the protocol. The device driver monitors the length of the RFL. Whenever an skbuff is "popped" to the device

driver, and the length of the RFL is found to have reached its original length of 32 buffers, the device driver simply frees the skbuff. The complete buffer lifecycle shown in figure 6 is summarized as follows:

1 - The APE25 consumes an RBH and its associated SKBUFF via the SRFL register.
2 - The APE25 produces a reassembled AAL5 frame onto RRL_n via the RRLn_LFDA register.
3 - The device driver's interrupt service routine consumes from RRL_n via the pointer ape->srrl[n]. The RBH is produced to free RBH list. The skbuff is "pushed" to protocol.
4 - The protocol returns the skbuff to device driver. The device driver consumes an RBH from free RBH list and produces the bound pair onto the RFL via the pointer ape->erfl.

4. IP over ATM

There are two standard approaches for providing IP service over ATM. The IETF's offering, Classical IP over ATM (CIP), is the simpler of the two and is defined in RFC 1577[Laub:94]. The ATM Forum's more complex proposal LAN Emulation (LANE) is defined in af-lane-0021.000[ATMF:95]. An in-depth comparison of the two approaches is well beyond the scope of this paper, and the reader is referred to [Kerch:97] or [Flowe:98].

4.1. CIP over PVCs

CIP relies upon an extension of the traditional ARP mechanism to provide IP address to ATM ("MAC") address mappings. Furthermore, CIP can be used in a strictly PVC environment with no need for signaling at all. Thus, in keeping with our philosophy of solving the easy problems before attempting the harder ones, we recommend starting out with a small PVC based network. In theory, any device driver that interoperates correctly with native Linux ATM should also support CIP.

In practice, we encountered three problems. First, we failed to either explicitly set (with ifconfig) the maximum transmission unit (MTU) for the link or to configure the device driver to support CIP's default MTU of 9180. This led to some relatively spectacular device driver failures because the APE25 *will* distribute oversize frames across multiple buffers whether the device driver is prepared to deal with them or not!

Next we discovered that CIP *never* returned skbuffs used in receive operations to our device driver's pop() function, ape25_free_rx_skb(). Our original strategy for replenishing the RFL was not nearly aggressive enough under CIP's "no return" policy, and the result was frequent exhaustion of the RFL and dropping of frames. This type of problem is particularly insidious because it does *not* cause system crashes. It simply degrades performance to a certain degree. We were able to detect it because we do count and report the number of times the APE25 generates an interrupt indicating that the RFL is empty. Once the problem was discovered it was easily fixed by increasing the initial length of the RFL and implementing a more aggressive replenishment strategy.

The final problem, which is still unresolved to a certain degree, is the transmit-related problem to which we have previously referred. When the TFD or TRQ resources that are required to initiate a transmission are not available, in our original design, the device driver sleeps until they are. However, when TCP/IP over ATM is being used sleeping is not an option. We found that when heavy UDP loads were imposed, the system log became filled with many lines of the ominous looking message:

```
AIEE... sleeping in interrupt handler.
```

We were able to conduct tests that transferred hundreds of megabytes of data without observing any related ill effects. Nevertheless, we elected to eliminate the problem by having the transmit routine simply drop the packet whenever the value of the kernel global variable intr_count was non-zero.

Since it was not clear why the device driver's transmit routine would be regularly invoked in the context of an interrupt handler, we traced through the IP code to try to better understand the problem. Along the normal send path for both TCP and UDP, is a call to dev_queue_xmit() (in usr/src/linux/net/core/dev.c). Function dev_queue_xmit() calls start_bh_atomic() (in include/linux/interrupt.h) which increments the variable intr_count. After the transmit is complete, control is returned to do_dev_queue_xmit() and intr_count is again decremented. That is, the device driver was *not* actually being called from an interrupt service routine after all.

Alan Cox [Cox:98] indicates that this mechanism is used for serialization and is necessary for the correct operation of IP. He also proposed a workaround in which the device driver could set a busy flag in the device structure pointed to by the skbuff. We understand this mechanism to work as follows:

```
netdev = skb->dev;        /* Recover pointer to struct device */
netdev->tbusy = 1;        /* Flag device as busy.             */
return(0);                /* Have the skbuff back for now..   */
```

To inform the upper level protocol that the supporting device is no longer busy the following code can be added to the transmit complete interrupt service routine.

```
if (netdev != 0)
{
     netdev->tbusy = 0;
     mark_bh(NET_BH);
     netdev = 0;
}
```

When we attempted to implement this mechanism, we found that UDP traffic incurred a packet drop each time we set the device busy flag. Analysis of the UDP transmit path failed to reveal any location at which a buffer could be requeued for later transmission. To avoid any possibility of skbuff leakage we returned to our original strategy of simply dropping frames whenever resources are unavailable and the value of intr_count is not zero. With our present allocation of TFDs and TRQ slots we find that the only way to induce drops is to impose a very heavy load of small UDP frames. The final version of the transmit sequence is shown below.

```
while ((tfd->tclnext == (struct tfdtype *)1)    ||
          (atm_rdsreg(ape, AAL_STATUS) & 1)     ||
          (ape->tbcount[lcndx] >= MAX_XBCOUNT))
{
      if (intr_count == 0)
      {
         interruptible_sleep_on(&ape->tfd_waitq);
      }
      else       /* Interrupt count => can't sleep    */
      {
      #if 0      /* Can't get packet put back to work */
          netdev = skb->dev;
          netdev->tbusy = 1;
          return(0);
      #else      /* but throwing it away is safe!     */
          if (vcc->pop == 0)
```

```
            dev_kfree_skb(skb, FREE_WRITE);
        else
            vcc->pop(vcc, skb);
    #endif
        return(0);
    }
    tfd = ape->sftfdl;
}
```

4.2. IP over SVC's: CIP and LAN emulation

As we said in the previous section, any device driver that interoperates correctly with CIP over PVCs should, in theory, also support CIP over SVCs or LANE. We found this to be almost true. The only modification made to the device driver was to add the code (shown in section 3.2) that reads the End System Identifier (ESI) from the APE25's NVRAM and stores it in the atmdev structure.

An interesting signaling related problem proved to be the main obstacle in successfully running CIP over SVCs and LANE. The external symptom was that CIP over SVCs would appear to start normally and work correctly for some time but then would cease working. The log file created by atmsigd was found to contain *many* blocks like the following:

```
atmsigd:KERNEL: TO KERNEL: as_indicate (0) for 0x0/0x1f3240 <184>
atmsigd:Q2931: SE vpi.vci=0.912
atmsigd:KERNEL: FROM KERNEL: as_reject for socket 0x8084898 (0x0/0x1f3240) in
stat
```

It is clear from the log that the switch was trying to assign vci numbers *well* outside the device driver's supported maximum of 63. The vci numbers assigned by the switch did roll over at 1023, but we needed them to roll over at 63. The documentation that was provided with the IBM 8285 switch also provided no hint as to how one might manually configure a switch port in this way.

It seemed reasonable to us that this information might be conveyed via ilmi, and the ilmi log did reveal that the switch was asking the Linux ilmid for the supported number of vpi and vci bits. However, it appears from the log that those names are not supported in ilmid's MIB. (The ASN.1 identifier `1.3.6.1.4.1.353.2.2.1.1.6.0` corresponds to the name atmInterfaceMaxActiveVciBits in the ATM Forum MIB.) This extract from the ilmi log shows the receipt of the query from the switch.

```
ilmid:IO: SNMP message received:
{ -- SEQUENCE --
    version 0,
    community '494c4d49'H  -- "ILMI" --,
    data get-request { -- SEQUENCE --
            request-id 75,
            error-status 0,
            error-index 0,
            variable-bindings { -- SEQUENCE OF --
                { -- SEQUENCE --
                    name {1 3 6 1 4 1 353 2 2 1 1 6 0},
                    value simple empty NULL
                },
                { -- SEQUENCE --
                    name {1 3 6 1 4 1 353 2 2 1 1 7 0},
                    value simple empty NULL
```

The response from ilmid is “no such name in my MIB.”

```
ilmid:IO: SNMP message sent:
{ -- SEQUENCE --
    version 0,
    community '494c4d49'H  -- "ILMI" --,
    data get-response { -- SEQUENCE --
            request-id 75,
            error-status 2,
            error-index 1,
            variable-bindings { -- SEQUENCE OF --
                { -- SEQUENCE --
                    name {1 3 6 1 4 1 353 2 2 1 1 6 0},
                    value simple empty NULL
                },
                { -- SEQUENCE --
                    name {1 3 6 1 4 1 353 2 2 1 1 7 0},
                    value simple empty NULL
```

We posted a description of this problem to the Linux ATM mailing list in January 1998 but received no suggestions on how to extend the ilmid MIB. Veli-Matti Junkkari at Telecom Finland provided us with a pointer to a source for upgrading the firmware in our switch. This upgrade permitted us to manually configure the maximum supported vci on each port. Once that was done, CIP over SVCs and LANE both functioned normally.

4.3. Performance comparison

To gain insight into the possible performance impact of using TCP/IP over ATM, we ran a series of tests designed to measure the maximum sustainable throughput achievable with the standalone driver, Almesberger’s native ATM, and TCP/IP over ATM. We conducted these tests on a pair of Pentium-based systems, a 180MHz P6 and a 100mhz P5.

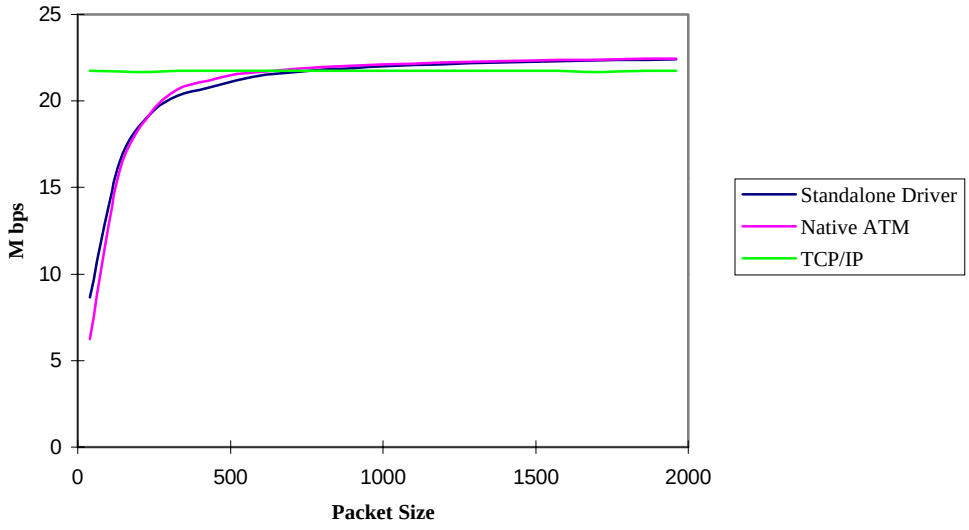

Figure 7: Sustainable throughput comparison.

Application packet size was varied from 40 bytes to 1960 bytes in steps of 80 bytes and in each test 50 Mbytes of data were transferred. No disk operations were involved as the sender simply retransmitted a single buffer until

50 Mbytes had been sent and the receiver used a single input buffer. For the standalone driver and native ATM tests, a sequence number was inserted by the sender and verified by the receiver to ensure that packets were not being dropped.

The APE25's physical layer transmission rate is 25.6 Mbps, but, due to cell overhead and the AAL5 CS trailer, transmission of an *N* byte packet from the application level requires transmission of at least 53 (*N* + 8) / 48 bytes by the hardware. Thus, maximum application layer bit rate is bounded by 25.6 x 48 *N* / 53 (*N* + 8) Mbps. The performance obtained with the three drivers is shown in figure 7. For a packet size of 1960 bytes, the maximum the application level bit rate is 23.09 Mbps. Both the standalone driver at 22.40 Mbps and native ATM at 22.45 Mbps achieved over 97% of the theoretical maximum.

For the TCP test, the interface was configured with a maximum transmission unit (MTU) of 2016. AAL5 frames carrying TCP packets must carry additional overhead of 48 bytes of TCP, IP, and LLC header data. This additional overhead bounds the maximum TCP application level bit rate at 22.5 Mbps. TCP's observed bit rate was nearly constant at an impressive 21.7 Mbps or 96.4% of the maximum possible. For small application level packets sizes, the performance benefits of TCP's transmit buffering are clearly evident in the graph.

5. Present status and future directions

The APE25 device driver in both its standalone and protocol-supporting forms has been stable since November 1997. Development has been carried out using Slackware distributions of the Linux 2.025 and 2.027 kernels and the 0.31 level of the Linux ATM protocol. Limited testing has also been conducted using Red Hat versions of the same kernels. The device driver interoperates with the Linux ATM protocol as a "bare-bones" SAR driver [Almes:96a] and presently provides neither QoS nor PHYS driver support to the protocol.

Extensive testing has been performed in a very limited environment, but minimal testing has been conducted outside that limited environment. The code will be made generally available under the GPL in May 1998. We have identified a reasonably small collection of experienced Linux developers with whom we will work closely in the beta testing phase of the distribution.

This device driver does *not* presently support 2.1 kernels nor levels of the Linux ATM protocol other than 0.31. In the past few months the protocol itself has been in a state of flux as its migration to 2.1 kernels proceeds. When this situation has stabilized, we will provide a 2.1 enabled device driver. This upgrade will also provide QoS support to the protocol. Making use of the APE25's scatter/gather capabilities in the reception of AAL5 frames is also under study. Significant improvements in the efficient use of buffer memory could accrue from this enhancement, particularly in a CIP environment.

In summary, we found that building an ATM device driver for Linux is an excellent way to extend one's knowledge of Linux kernel programming, Linux networking in general, and ATM networking in particular. Given adequate documentation, an experienced system programmer should be able to construct a reasonably functional device driver within a time period of a few weeks.

References:

[Laub:94] M. Laubach, “RFC 1577: Classical IP and ARP over ATM”, January 1994.

[ATMF: 95] ATM Forum, “LAN Emulation over ATM - Version 1.0’’, Document af-lane-0021.000, ATM Forum, January 1995.

[Almes:96a] W. Almesberger, “Linux ATM Device Driver Interface Draft”, ftp://lrcftp.epfl.ch/pub/linux/atm/docs, Feb. 1996.

[Almes:96b] W. Almesberger, “Linux ATM API Draft” ftp://lrcftp.epfl.ch/pub/linux/atm/docs, July 1996.

[Marko:96] K. Marko, “Implementation of LAN Emulation Over ATM in Linux”, M.S. Thesis, Dept. of Information Technology, Tampere University, Tampere Fi., Oct. 1996.

[IBM:96] IBM Corp., “ATM Protocol Engine Functional Specification 3.0”, IBM Networking Systems, RTP NC, Aug. 1996.

[Tanen:96] A. Tanenbaum, *Computer Networks,* Prentice-Hall, Upper Saddle River, NJ, 1996.

[Kerch:97] B. Kercheval, *TCP/IP Over ATM,* Prentice-Hall, Upper Saddle River, NJ, 1997.

[Stall:98] W. Stallings, *High-Speed Networks: TCP/IP and ATM Design Principles*, Prentice-Hall, Upper Saddle River, NJ, 1998.

[Geist:98] R. Geist and J. Westall, “Bringing the high end to the low end: High performance device drivers for Linux”, In *Proceedings of the 36th Annual ACM Southeast Conference,* pp 251-260, March 1998.

[Cox:98] A. Cox, Linux-ATM Mailing List, linux-atm@lrc.di.epfl.ch, March 19, 1998.

[Flowe:98] A. Flower, “System Integration and Performance Testing of IP Over ATM”, M.S. Research Paper, Dept. of Computer Science, Clemson Univ., Clemson, SC, April 1998.