Report on the Development, Structure and Performance of ATM Device drivers for the Personal Computer operating systems Windows NT, and Linux

Richard Black

28th September 1995

Chapter 1

Introduction

This document describes the structure and and implementation of the device driver for the EATM ATM network interface card.

The first phase of the project required the development of a device driver for the Windows NT operating system. This took a great deal of effort over many months. The relevant features of Windows NT, and the process that lead to the development of the driver are described in chapter2.

As a result of the negative experience with Windows NT, particularly the provisional performance measurements, it was felt that it would be interesting to compare Windows NT with a device driver for another operating system on identical hardware. The LINUX system was selected for this comparison. Further details about Linux and a report on the implementation of the device driver may be found in chapter 3.

The general environment used for testing and measuring the device drivers is described in chapter 4.

Chapter 5 reports on the comparative ATM performance of the device driver on identical hardware. The underlying differences between Windows NT and LINUX are described.

Finally appendices A and B give some sample code for accessing AAL5 PDUs from a user program on Windows NT and LINUX respectively.

Chapter 2

An EATM driver for Windows NT

2.1 Introduction

The author has previously implemented an ATM protocol family in other operating systems including UNIX [**Black94a**]. The Windows NT operating system was expected to form a difficult platform for the implementation of a new network protocol due to Microsoft's history of failing to adopt concepts proven by the research community, and this indeed turned out to be the case.

The documentation available for NT is very poor, certainly not as accurate or as well cross-referenced as the Unix man pages. Contrary to initial expectations, NT source code was not available.

The implementation method adopted was chosen with a few months experience and research into the internal structure of NT, it may turn out retrospectively that there is a better method. This potentially partial knowledge applies to all of the information about NT within this document.

2.2 Windows NT Features

This section provides an overview of some of the features of Windows NT which impact the design and implementation of the ATM protocol.

2.2.1 Windows NT Network Structure

Some significant time was spent investigating the existing networking code within NT, specifically the NDIS 3.0 and miniport driver structure.

The purpose of the NDIS and miniport interfaces (and all of the other device driver interfaces within NT) is to provide an interface which allows an OEM to port a pre-existing sample or library device driver to their hardware. The underlying assumption is that Microsoft have already come across all the types of devices that could possibly be attached to a personal computer and that (likewise) they have written library code for manipulating all of the chip sets that could be found in a PC. It is considered to be the OEM's job merely to link together the pre-supplied components which may be found at subtly different bus addresses or interrupt lines on their systems. As a result the interface is not well suited to the introduction of a new type of device[1].

The consideration of the NDIS interface included examination of a sample device driver for an Ethernet chip. Despite the use of library code discussed in the paragraph above this driver turned out to be extremely bloated. For comparison, a single sample Ethernet driver in NT came to in excess of 10,000 lines of code; for all possible bus attachments for an equivalent chip on DEC workstations, the UNIX driver is about 3,000 lines.

[1]The provision of an ATM device driver by Microsoft will not solve the general problem nor will it allow experimentation with ATM protocol stacks.

The publicly available interface to the networking code assumes that all network interfaces are of two overall classes: These are LAN interfaces and WAN interfaces. NT assumes that LAN interfaces are packet based, that they are all capable of multi-protocol encapsulation, and that they are all connection-less. A LAN interfaces must be one of three possible types. Their interface with the rest of the system is dictated by what type of LAN interface they are. The three types are "NdisEth" for Ethernet drivers, "NdisFddi" for FDDI interfaces and "NdisTr" for Token Ring interfaces.

A WAN interface, on the other hand, must be one of two types: Serial lines on which the NT system will enforce PPP access methods, or ISDN lines on which NT will perform its own signalling.

None of these can be accessed directly by user processes but only by higher level NT protocols (such as IPX and Netbios). No other type of LAN or WAN interface is supported and it is not possible to provide a generic packet interface (as one would with `struct ifnet` in Unix).

The use of the NT networking code for this project was therefore rejected on two principle grounds:

1. It would be difficult to provide an effective implementation of LAN emulation given the above assumptions.
2. There could by no provision of direct access to the ATM protocol from a user space process, a requirement of the project.

It is interesting to note that the Fore systems ATM network interface driver does not provide the Fore systems standard ATM API but provides only LAN emulation.

2.2.2 Windows NT device structure

As described in the previous section the implementors of NT used a rather unhelpful set of assumptions when they designed the network device driver interfaces. This extends to other devices in general. There are different customised interfaces (and sample device driver code) for disk drivers, for CD ROM drivers, for parallel ports, for sound cards, for video cards, and most other imagined devices. Unfortunately there is no (documented) generic device interface.

After a request on the internet newsgroup `comp.os.ms-windows.nt.misc` some sample code for a statistics counting device driver which operated as a generic device was obtained. By reading the documentation for the functions used by this device driver it was possible to understand how to write a general purpose device driver.

The general purpose device driver thus created is most similar in concept to the Unix character special device (e.g. as found in `/dev`) but it has a much more complicated structure and convoluted initialisation than one would find on Unix. There are also a lot of similarities with serial line device drivers for NT since they are also accessed asynchronously and with unformatted data.

2.3 Overall structure

The overall structure of the ATM protocol family was originally planned to be in the general form of [**Black94a**] as shown in figure 2.1, but experience with NT forced this design to be altered. Instead the structure is now intended to be as shown in figure 2.2. This is because the kernel device driver is not in a position to relay the signalling requirements to a signalling system since the NT system makes no provision for connection-oriented information in the driver, and because the interfaces necessarily provided by the kernel device driver are too low level.

The current design calls for a Dynamic Link Library (DLL) above the level of the kernel. This

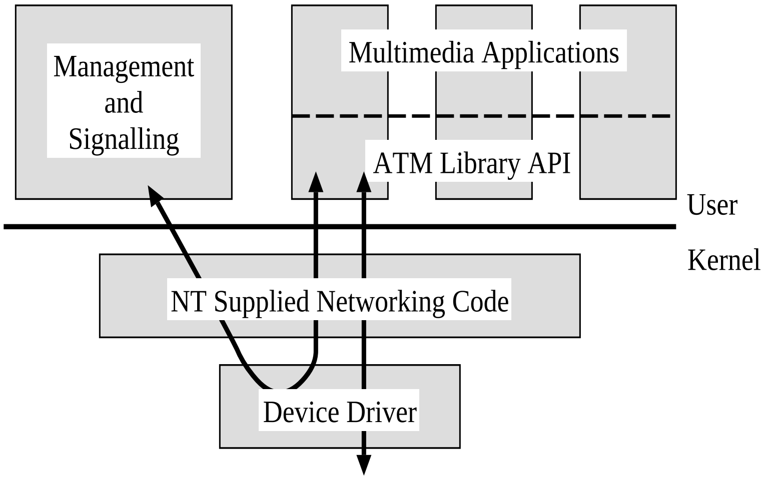


Figure 2.1: Original Overall Structure Design

will interact between the user processes and the kernel to provide a more friendly API to user space programmers as well as providing connection-oriented security (see the further discussion of security below). The DLL will, therefore, also contain the signalling component of the system. User processes will make accesses to the DLL through an ATM customised interface which will check these requests and pass them on to the kernel in the appropriate manner.

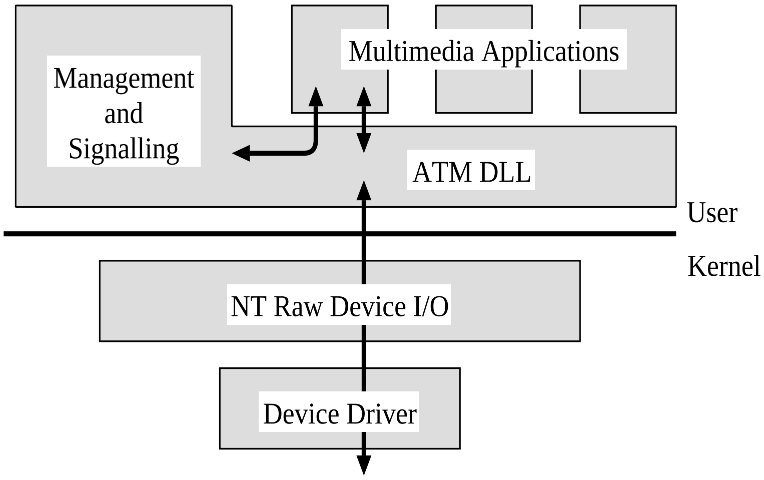


Figure 2.2: Revised Overall Structure Design

2.4 Driver Implementation Details

This section describes in detail the structure of the device driver as it is currently implemented. The device driver is still categorised as being in development (though stable) rather than as being complete.

The device driver is designed with the potential of supporting multiple ATM device interfaces, though at present a number of the initialisation and naming issues mean that only one interface is supported. Support for multiple interfaces should not be excessively difficult to add at a later date if required.

2.4.1 Initialisation

The device driver is initialised during the boot process. This is configured by manually editing the system configuration database known as the *registry*. For a product system this function would be carried out by the vendor's GUI installation program. For a research environment the construction of such an installation program is unnecessary, and no allocation has been made for that in the research proposal.

2.4.1.1 Early initialisation

The driver's initialisation code first executes a breakpoint instruction. This is followed by a test of a static flag which is compiled in with value "false". If this value is "true" then the device driver exits without performing any more operations. This code sequence is very important for the turnaround cycle. If a bug exists in the device driver which would cause it to crash during initialisation then it may be necessary to recover the system using the emergency repair procedure. That procedure takes about 3 or 4 hours. Instead, if the device driver is known to be faulty, the flag can be set to "true" at the initial breakpoint, then the system will boot without the ATM device driver, and the bug can be corrected. This allows a much faster turnaround.

The driver then conducts a scan for the ATM network interface card. In theory the NT interface for card location supports a large number of different busses containing many different slots. The ATM device driver therefore iterates over many busses and slots and asks the NT kernel Hardware Abstraction Layer to return information about the card (if any) present in that slot. For an EISA bus, the NT kernel in turn gets this information from the CMOS RAM which was initialised by the EISA system configuration utility from the device's .cfg file. If the information returned indicates that the EISA Board Identification number is that of the ATM interface card then the ATM device driver ceases its search and proceeds to the next stage of initialisation.

2.4.1.2 Initialising NT resources

An NT DEVICE_OBJECT is then created. This object can include an area of memory designed for device specific usage. Unfortunately the NT kernel contains a bug which means that the area required by the device driver is not taken into account and the NT kernel will corrupt the system heap and the entire system will crash in a non-deterministic way shortly afterwards. Discovering this sort of bug, and developing workarounds, is the key reason for the flag described earlier.

The workaround for this particular problem involved reporting that the driver required no additional bytes of storage, and allocating the required memory directly in the driver. A pointer to this field was then placed in the internal field in the DEVICE_OBJECT which is reserved for use by NT for controlling the driver's memory. When destroying the DEVICE_OBJECT the driver takes care to put this field back as it was (and freeing the driver memory separately) before instructing NT to delete the DEVICE_OBJECT.

The driver then selects "direct I/O" operation. This causes the NT I/O system to lock down pages which are used as I/O buffers for this device so that the data is rapidly available. If this is not selected, the NT system allows memory areas involved in I/O to be paged out and accesses must be made using tiny buffers which are copied to and from the user memory. Selecting "direct I/O" is important for performance and for ease of use.

A "symbolic link" is then created. This has in fact nothing to do with symbolic links as such, but is the NT term for exporting a device into the Win32 name-space. Applications other than the kernel may only access things which are named in the Win32 name-space. The impact of name-spaces is also discussed in section 2.6.

The driver is now able to acquire hardware resources on behalf of the device object. Initially this code used the HalAssignSlotResources function because the documentation stated that this function deduced the requirements of the device in the slot from the configuration data. After a long time it was discovered that this function does not actually work but causes general system corruption and random crashes.

On a second attempt the device driver was coded to unmarshall the EISA configuration information by hand and then remarshall it into the completely different format required by the IoAssignResources function. However it was discovered that this function also has too many bugs to be relied upon, and caused frequent crashes.

Following the acquisition of sample source from the internet, a third (successful) mechanism was adopted. This is what the driver now does:

1. The device driver allocates a system interrupt vector based on the bus and slot number and the interrupt number extracted from the EISA configuration information.
2. The address of the card on the EISA bus is converted from a bus-relative address to a system physical address.

The driver then marshalls the information resulting from these operations into yet another function specific format and calls the IoReportUsage function. This function assigns the resources and updates the runtime system database. It appears to be reliable.

The driver now deals with initialising the card. This is done in two stages. First the driver maps the system physical address determined in (2) above into the supervisor virtual address space. This allows access to the card. This address is then stored in the device object. The card can now be initialised. The driver configures the NT system with a "delayed procedure call" entry point for the device driver. This is then used with the interrupt vector and information found earlier to connect an interrupt handling routine to the card's interrupt vector. As soon as this call has completed there is the potential for the system to take an interrupt from the card. This is why the initialisation is done in this order (specifically the interrupt routine can access the device using the mapped pointer stored in the device object in order to clear the interrupt again).

2.4.1.3 Initialising I/O mechanisms

The second phase of initialisation is now complete and the driver moves on to initialisation of the I/O interaction. This involves inserting various function pointers into the driver and device object structures. This is so that the device driver may be unloaded at system shutdown, and also provides handlers for the appropriate I/O functions that will be needed by clients of the device driver.

2.4.2 ATM Data Path

As a result of the initialisation code, the NT system has a device name for the ATM hardware interface. Operations on this device by user programs are converted by the NT kernel into calls on methods in the device driver code.

The device driver supports AAL5 only. This is all that would be required for multimedia or conventional data applications. Support for additional AAL types (e.g. AAL0 for OAM cells) could be added at a later date if necessary.

2.4.2.1 Client state information

As discussed already in section 2.2.2, the NT system has a very peculiar attitude to the organisation of devices. This has important bearing on the information that the driver must keep on behalf of each user (for each virtual circuit).

The NT system either regards a driver as either providing a file (of device type), or a file-system. The latter allows multiple names to be handled by the driver, but would be enormously heavy-weight and impractical for this application. For the former, the system conceives of no possible per-client state requirement; the file contains some number of bytes which may be accessed by any client.[2]

In order to encode the required client state, the device driver uses the only piece of meta-information which is available to it on both reads and writes: the file position. When a receive or transmit is wanted on a particular VCI, the client performs an lseek[3] to an encoding of the VCI required and then performs the I/O operation. The device driver uses the file positional information in each read/write request to determine which VCI to use for the operation.

2.4.2.2 Transmit Operation

Transmission of AAL5 requests is performed synchronously by the device driver. This decision is due to experience on other platforms which showed that for a programmed I/O 100Mbit/sec TAXI interface, operating systems are not able to keep with the line rate. This is even more likely to be the case in a system as poor as NT, and with the low bandwidth of the EISA bus.

When the driver is invoked, it checks the arguments to the operation for validity: length, alignment, and that the correct AAL5 padding is supplied. The requirement on the user application to supply correct padding is not onerous and greatly reduces the complexity of special case processing within the driver.

The driver then takes out a lock on the operations of the card (using a concept designed for controlling interrupts) which protects against simultaneous access by other CPUs (in a multiprocessor system), other threads, and the action of the interrupt handler. The AAL5 PDU is then written to the interface.

2.4.2.3 Receive Operation

The implementation of receive is always more difficult than transmit due to the fundamentally asynchronous nature of the operation.

A considerable time was spent examining the documentation for the NT kernel and the Win32 subsystem to discover what buffering strategies and asynchronous mechanisms were supported. As a result, the mechanism adopted is substantially different from that used in previous ATM device driver implementation such as Unix [**Black94a**], and Wanda [**Dixon91**] and in certain ways is more similar to the Rbuf mechanism of [**Black94b**].

The design choice requires user processes to issue read request(s) asynchronously, providing buffer memory to the device driver in advance for data to arrive in the future. The reasons for this are:

- The NT system has various constraints on which areas on memory may be allocated or

[2] In some ways this is reminiscent of a problem with the packet filter in Unix systems, although in those systems the problem is solved relatively easily using minor device numbers. See also the system used in Linux described in chapter 3.

[3] On NT this standard C function is not supported but equivalent functionality is provided through a rather baroque interface.

referenced in various situations and at varying interrupt levels. This would have added complexity to buffer management within the device driver.

- The use of kernel buffer memory would entail an additional copy between the kernel buffers and the user space memory, reducing performance.
- The use of user space buffer memory allows much better flow control in the system. In the event of overload, data arriving will be discarded without the overhead of copying into kernel buffers and queueing for a user space application.
- When using kernel buffers, the kernel does not know in advance the expected length of PDUs on any particular VCI. This in turn would require it to support the chaining together of multiple kernel buffers; it is too wasteful to allocate 64K buffers for every PDU.
- Using kernel buffers the device driver has no way of knowing when to garbage collect memory for redundant VCIs. This is due to the lack of client state discussed in 2.4.2.1 above. It could lead to large amounts of memory being permanently allocated for VCIs belonging to processes which are no longer extant.

When a user process wishes to receive data in the future it issues a Win32 overlapped I/O read call on the file handle[4] specifying the position in the file (i.e. the VCI number) and the buffer to be used. This buffer must be aligned correctly and padded to the correct length.

When the device driver receives this request it calls a function to map the user's buffer into supervisor virtual addresses. This function also pins these pages into physical memory so they are guaranteed not to be paged out and can therefore be safely accessed from the receive interrupt routine. The buffer is then queued on a per-VCI queue[5] within the device driver's internal structure. The data stored includes various information to ease the handling of the buffer during the performance critical receive interrupt processing.

Before the device driver can indicate to the NT kernel that the I/O request is to be considered as pending, it must concern itself with the handling of the request in the case that the I/O request is cancelled for some reason (such as the process exiting or being killed). This is done by registering a cancel routine for the request (which must be done while holding a special type of system lock). This ensures that the kernel will give the driver an opportunity to remove the request from the VCI's queue before the memory buffer is deallocated. When these operations are all complete the device driver can return.

The cancel handling code is entered with the special cancel system lock held and a pointer to the I/O request that is being cancelled. The cancel code then acquires the driver's interrupt lock to ensure safe access to the VCI queue structures, removes the entry, and releases the lock. Finally it releases the system cancel lock and calls back into the I/O system to declare the I/O request as having been completed. This is quite a peculiar operating system interface but appears to function correctly.

2.4.2.4 Interrupt handling

When an interrupt occurs the NT system calls the interrupt stub declared during initialisation. Under NT this stub is not permitted to actually perform any significant processing, instead it must simply disable the interrupt condition and request a "delayed procedure call" (DPC).

[4]Even though the functions used are different from those used for synchronous I/O, a file handle cannot be used for both operations; this necessitates opening the ATM device twice, once for receive and once for transmit.

[5]The number of VCIs supported is limited by the number of queues. This is a compile time option for the device driver.

The delayed procedure call must ensure that access to the card is atomic with respect to other driver activity, so it acquires the interrupt lock described in section 2.4.2.2, calling the receive processing code.

The receive processing code operates as a state machine operating in a loop whilst there are cells in the buffer memory on the interface card. The header is used to determine which VCI queue to use. The state of the current PDU being reassembled is then loaded from that queue entry and the cell data copied to the destination. A fast code path exists for handling multiple cells from the same PDU which are adjacent in the interface. Experience from previous work shows that this is an important performance optimisation. As each PDU is completed, and if the length and CRC are correct, it is placed on a special internal queue of completed requests.

This internal queue is very important because the NT system is structured in such a way that the receive code running, as it is, with at least one interrupt lock held, cannot make certain calls into the I/O system. One of these calls is the call required to tell NT that a previously pending I/O request is now complete. Instead, when the receive handling code finally returns to the DPC routine and the interrupt lock is freed, the DPC routine then completes all the I/O requests on this internal queue.

2.4.2.5 Raw device access

The kernel device driver also supports a method for allowing a client to make direct single word read or write cycles to the hardware registers on the interface. This interface existed originally to aid in debugging, and is likely to be removed in due course. One possible use for it might be for an OAM function which wished to query the error conditions recorded by the card for the purpose of monitoring the link quality. Such an interface would have to be carefully controlled as it has the potential for causing the hardware to lock up.

2.5 Development Environment

The development environment consists of two 486-based PCs in the Laboratory's NT administration domain, plus the author's Unix workstation. The two PCs belong to the project, the workstation does not. One of the PCs has an EISA bus with the EATM card. It runs the "checked build" version of NT which is one with additional run-time assertions included. The other (running the normal or "free build") runs a debugger. The debugger and the experimental machine communicate over a dedicated serial link. This allows the use of breakpoints and single stepping within the kernel of the experimental machine.

The experimental (target) machine has the Microsoft Visual C++ compiler version 2.0 installed on it together with the Software Development Kit (SDK), which is necessary to do any development whatsoever, and the Driver Development Kit (DDK) which is necessary for kernel work. The other PC has only the SDK which is required for the debugger.

The workstation is used for editing the source code and other computer environment tasks using established and robust software. The NT CD ROMs are also mounted on other Unix workstations so that they can be subject to standard programming tools such as find and grep which do not exist for NT.

The Unix distributed NFS file-space of the Laboratory can be mounted on NT using the SAMBA software which runs on a central Unix server. This free software is superior to all the commercial NT NFS access products and, because it runs on a stable Unix platform, is highly reliable. SAMBA works by deciphering the requests of the NetBui protocol and performing standard Unix file access requests to obtain the data which is then returned to the NT machine. NT machines are not able to export their file-systems using standard protocols such as NFS (they can be accessed from other

NT machines).

For robustness, the device driver, test programs, and compiler code are stored on the Unix file-system. The finished driver is installed on the experimental machine prior to test. Thus the project's work is all automatically backed up and managed by normal Laboratory staff.

The NT system is currently running the device driver development kernel (known as the "checked build") rather than the normal production kernel (known as the "free build"). This is expected to have a minor detrimental effect on the performance of the system.

2.6 DLL Design

The overall design mentioned in section 2.3 called for a DLL to provide the signalling and user API interface between the user programs and the kernel device driver. Although no such DLL has yet been implemented this section describes some of the issues involved.

The kernel device driver is not able to operate in a connection-oriented manner or relay signalling requests from the clients to a signalling manager. For security reasons, the interface provided directly by the kernel device driver should not be accessible to applications in general, since it would allow them to circumvent the signalling and management function.[6]

Further, the actual interface described in section 2.4.2 is unpalatable to application programmers. The DLL would be responsible for providing a uniform secure connection oriented application interface. Behind this interface it would check the connection being used, hide the requirement for multiple file descriptors, and perform any signalling as required.

It was originally intended to implement the signalling function within the Posix subsystem, however this has been rejected on the discovery that NT appears to implement a very strict interpretation of the Posix standards. For example, since the standards only specify access to a file system, the NT system does not appear to permit Posix programs to access devices or other services within the machine. In particular there appears to be no way of exporting a device in the Posix name-space as mentioned in section 2.4.1.2. This effectively rules out its use as a base for managing other parts of the system.

[6] It is not currently known how to restrict access to the device to a DLL in NT, but it is known that this can be done and it is expected that this security concern will be addressed later in the project when further experience with NT has been obtained.

Chapter 3

An EATM driver for Linux

3.1 Introduction

Linux is now the second most commonly used of all Unix variants. The core of the operating system is freely available (including source code) and the general Unix utilities are also free (using the gnu project utilities). Commercial applications may be used with Linux, and Linux is now a supported platform for many commercial applications. The most recent versions of Linux support the Intel ELF portable executable format for moving application binaries between it and other systems.

Unlike NT, the documentation available for Linux is extremely effective. As well as the manual pages, there are many documents available freely on the internet which describe the operation of the system in a *technically accurate* manner. Whilst perhaps less suitable for non-technical managers, this sort of documentation is of course much more suitable for programmers. Additionally, the author was able to make extensive reference to the commented source code for the kernel (including other device drivers) which is provided with every distribution.[1] Furthermore, the authors of the various components of the system have their email addresses listed in the credits, and can be contacted directly for rapid (and free) technical support.

Since Linux was known to be a fairly simple generic Unix, and since it was desired to implement a simple AAL5 device driver very rapidly for comparison purposes, the decision was made to use a simple character device providing AAL5 access only, and avoid integration with the linux networking code. There is a separate linux project to add ATM support to the networking code in general. Details of this project may be found at `http://lrcwww.epfl.ch/linux-atm/`.

3.2 Linux character device structure

A character-special device driver is constructed as a set of methods for handling requests for a small number of operations on files. These are seek, read, write, read-directory, select, ioctl, mmap, open and release. Not all of these operations are appropriate for all drivers, in this case the methods provided are read, write, select, ioctl, open, and release.

In Unix, by convention, device special files (created with mknod specifying a character or block indication together with major and minor device numbers) are placed in the /dev directory, and given a name which identifies in a human-comprehensible way, the associated device driver. It is actually the major number of the special file which determines which device driver will be called when the file is opened. The minor number is available to the driver for internal discrimination.

When file read or write operations are performed the linux kernel checks the arguments for being valid addresses before calling the drivers routine. Special routines (which may of course page-fault on paged-out data) are provided for copying to and from the kernel memory. These routines

[1] It ought to be remarked that, though slightly Intel-centric, the quality and clarity of the code surpasses that of any of the various commercial Unix versions that the author has seen.

cannot be accessed in interrupt routines since interrupt handling code should not page-fault.

3.3 Driver Implementation Details

This section describes in detail the structure of the device driver as it is currently implemented. Due to the better reliability of the Linux operating system, compared with NT, the Linux driver was actually used for tracking down and fixing the final problems with the hardware.

The device driver is designed with the potential of supporting multiple ATM device interfaces, though at present only a single interface is supported. Support for multiple interfaces would be easy to add at a later date if required.

3.3.1 Initialisation

The device driver is compiled and linked into the kernel by modifying the kernel compilation configuration. The driver's initialisation routine is called during system initialisation.

In linux drivers are permitted to issue I/O operations directly to the bus; also the kernel virtual addresses are arranged to be identical to bus physical addresses so devices may be accessed directly.

The driver scans the EISA bus looking for an EATM card, by reading the EISA ID bytes for each slot. If the EATM card is found then the driver interrogates the ESIC chip [**Hitatchi91**] which has been already been programmed by the BIOS as a result of the EISA configuration file. This information is guaranteed to be correct since it is read directly from the control registers on the device.

The driver resets the card and disables its interrupt conditions and then registers its interrupt handler with the kernel specifying the interrupt number read from the configuration. It also initialises a regular timer for 1Hz to function as the regular status monitoring watch routine.

Finally the initialisation routine registers the major device number of the EATM driver with the kernel passing a pointer to the set of methods to be used for servicing requests to files of that type.

3.3.2 Client State Information

The driver must keep some state information on a per-client basis to ensure that exclusive access is provided to VCIs. This is done using the minor device numbers. There are many different files in `/dev`, one for each VCI which can be supported by the device driver. The minor number of the character special file is the same as the VCI that will be used if that file is opened.

Since the inode numbers differ this allows the device driver to distinguish clients. The driver will not permit the same minor number to be opened more than once, guaranteeing exclusive access. A client is still free to replicate its file handle and pass it to children or other processes in the usual unix manner if it wishes.

This use of minor numbers is only a temporary measure because no management or signalling software has been written. If such software were present, then it would be responsible for allocating the VCIs and passing the file descriptor to the other process.

Like the NT driver it was decided to avoid copying between the user memory and kernel memory if at all possible. In the linux system this is performed by arranging for the client and the device driver to share a shared memory segment. Exactly one such segment must be used by each client.

The shared memory segment must be allocated and made resident by the user. The user then performs an ioctl on the file descriptor to inform the EATM device driver that it wishes to use that segment. The driver checks that the segment is owned by that user and that it is resident. It will then mark it as locked down so that it will remain permanently resident until deleted. This segment is then used by all the date transmit and receive operations.

3.3.3 ATM Data Path

Both data path operations are specified by the user using an array of offset and length pairs which refer to the shared memory segment. These are converted by the driver into a (larger) array of similar pointer and length pairs where the differences between user virtual and kernel physical addresses have been taken into account (including breaking at page boundaries). This is following the design of [**Black94b**].

The main difference from the Windows NT driver is that there is no contiguous kernel virtual mapping for the user memory and so the driver must deal with the various discontiguous regions. Once this cost has been accepted it is then easy to allow the user space program a certain amount of flexibility in its requests.

3.3.3.1 Transmit Operation

Like the NT driver, transmission of AAL5 packets is performed synchronously by the device driver. The user program calls a write system call where the data for the write is actually a set of the offset length pairs. If these are converted to kernel addresses successfully, then the device driver knows that it will not fault in accessing the data in the shared memory.

The driver disables interrupts for the actual transmission of the data to ensure no parallel access if cells were to arrive, and copies the data to the interface card.

3.3.3.2 Receive Operations

Like the NT driver above, the driver is only prepared to receive data into buffer memory already prepared by the user.

The offset and length pairs required for the buffers are given to the device driver using an ioctl. These are stored in a list by the device driver for subsequent use by the interrupt routine.

The device driver supports the unix select function, so user processes can determine when data has arrived. The user may then read from the file descriptor. Reading returns the information about the packet at the head of the received list for that VCI. This gives the actual length received and a pointer to the information which was given to the kernel in the corresponding ioctl. This allows the user to determine where it had requested that the driver put the data.

Unlike the NT system where a complex cancellation mechanism was required, handling of aborted processes is simple in linux because the file descriptor will be closed by the kernel. If this happens the driver will free its reference to the shared memory segment, and the lists of pending and completed receive requests.

3.3.3.3 Interrupt handling

Again in this case the receive state machine is complicated by the requirement to deal with discontiguous memory areas. The actual code for copying data in from the interface is otherwise identical to the Windows NT driver.

No internal queue is required for completed requests between the interrupt handling code and the device driver receive code because linux is structured to allow device drivers to call scheduling operations from within interrupt handling code.

3.3.3.4 Other access

The kernel device driver also supports a number of other ioctls for accessing various statistics and internal configurable maximum values. As a special case, minor device number (which would have been the invalid VCI 0) is not valid for transmit or receive, and exclusive access is not enforced. This allows an arbitrary number of client to open this file for accessing statistical information.

Since the Unix kernel allows easy access to security information, the ioctl for resetting the interface card is protected from ordinary users and is only carried out if requested by the super-user.

3.4 Development Environment

The development environment for linux consists of a single EISA based 486 PC which was the same machine used for the NT development. The linux boot loader is used to switch between the two operating systems as it can easily support many different systems whereas the NT system uses closed technology and does not support linux.

All of the operation of the linux machine was controlled by logging in remotely using the standard protocols such as TCP/IP and X11. This allowed full control over the machine from the author's workstation.

All of the kernel files which were modified from the standard linux distribution were stored on the computer laboratories general unix file system and access using NFS via symbolic links placed in the linux compilation tree. These are more useful features not supported by NT.

The linux boot loader mechanism supports a very fast method for switching between multiple kernel images. This makes recovery from error much faster than NT. In fact a typical linux kernel compilation and reboot time is faster than simply logging in on NT. Additionally linux allows the use of printf[2] in the kernel; the messages are written to the system log and can be viewed at leisure.

All of these features make linux a much more friendly environment for developing a device driver. Together with its superior structure and reliability this accounts for the fact that the time taken to write an initial driver which supported transmission only took three *days* using linux, and about three *months* using NT.

[2]Actually called printk.

Chapter 4

Test Environment

Both drivers are instrumented. By recording the number of times certain events take place this allows statistics monitoring code to determine the circumstances under which packet loss takes place. This is necessary to determine the costs associated to make performance measurements.

The drivers also include link level sceptic code for handling faults on the line and recovering the card in the presence of errors.

Test software has been written for both NT and Linux. It can be used either as a receiver or a transmitter. Multiple instantiations of this software may be run at the same time. The software is used to transmit various data packets for testing purposes and also to print various statistics of hardware and software events within the driver for diagnostic and performance purposes.

The current test environment includes a bidirectional co-axial link from the EATM card to one of the port controllers [**Hayter94a**, **Hayter94b**] on a Fairisle ATM Switch [**Black94c**]. This equipment is provided by the Computer Laboratory. The port controller runs some specially constructed software which means that all of the cells sent by the test machine are returned to it. Additionally this software allows the tester to monitor the cells on the link.

For other testing, such as stability under load when receiving, the PC is attached to a port on a Fore Systems switch. This allows video from an audio visual capture unit to be routed to the PC. This unit is the Fore Systems AVA200. The control software for this unit is run on the author's workstation and provides very fine control over the operation of the unit and the bandwidth of the video streams.

Chapter 5

Comparative Performance

Performance comparisons are considered in two section dealing with transmit performance and receive performance.

5.1 Transmit performance

Transmit performance is measured by using a test program which sends a large number of AAL5 PDUs of a particular size and uses the systems time to measure the cost for sending that data. This is then repeated (automatically) for various sizes of packets.

The NT system provides approximately millisecond accuracy and there is no convenient way of checking whether the clock is drifting or not. The linux system provides approximately microsecond accuracy and since it runs the standard internet time protocol software (`xntpd`) it can be checked remotely to determine whether it was drifting significantly. The PC is on the same Ethernet as the Computer Laboratory's two highly accurate stratum 1 time servers which in turn receive time from the Rugby atomic clock radio signal. The linux system was observed to remain highly accurate during the transmit experiment.

In any case, each packet train was configured to take a substantial number of seconds to complete, thus making for an accurate reading on both operating systems.

A graph of the transmit bandwidth achievable for different packet sizes for both NT and Linux is shown in figure 5.1.

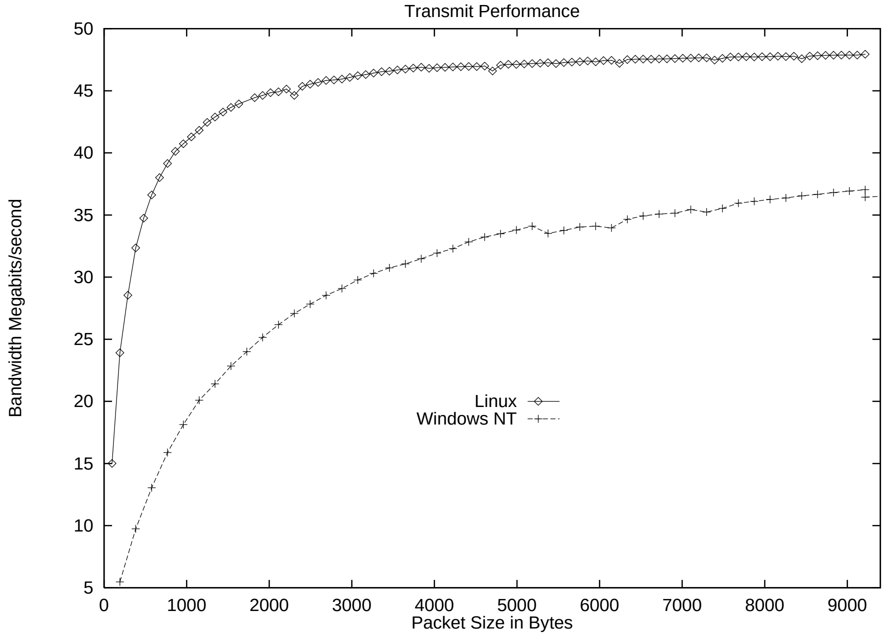
Figure 5.1 shows the same information presented graphically in two different formats. The first format shows the bandwidth achievable on transmit for various different packet sizes. It can be seen that linux does much better than NT, especially for packets at sizes likely to be appropriate for multimedia streams.

The second graph shows this information presented in reciprocal, showing the time taken to transmit a single packet for various sizes. This graph shows two straight lines, from which the important details of the two systems can be determined. These are the driver sustained transmission rate (when the driver is copying data to the EATM interface as fast as possible) which is the slope of the line, and the operating system per-packet overhead which is the intercept of the line. These are shown in table 5.1.

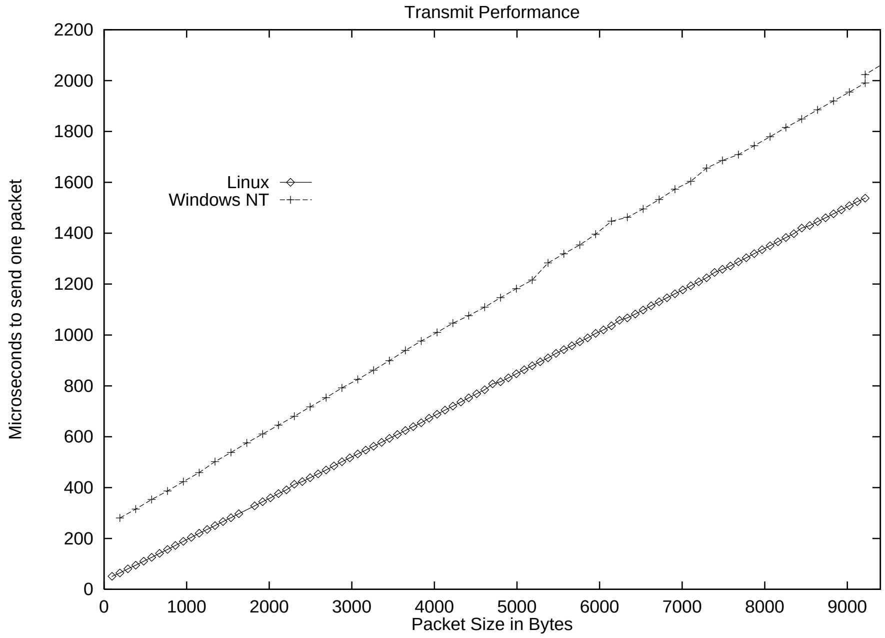
	Windows NT	Linux
Per-packet overhead	250μs	40μs
Driver copy rate	42 Mb/s	49 Mb/s

Table 5.1: Transmit Parameters

The principal effect on the performance difference is the per-packet overhead. It can be seen that Windows NT is much slower than linux and this accounts for the poor performance on NT. If the



(a) Bandwidth (direct)



(b) Elapsed Time (reciprocal)

Figure 5.1: Comparative Transmit Performance.

EATM interface / EISA bus combination had been capable of operating at the full 100Mb/s line rate then the *difference* in this per-packet overhead would account for 50 cells or $2\frac{1}{2}$ Kbytes.

An additional minor factor is the difference in the driver copy rate. Even though this inner loop in the device driver is very slightly more complicated in linux due to the potential for non-contiguous addresses, the linux performance number is higher. This is believed to be due to the (free) gnu C compiler (gcc) producing better code than the (expensive) Microsoft Visual C++ V2.0 compiler.

5.2 Receive Performance

Experimenting with the system under substantial receive load uncovered a number of minor bugs in the EATM card hardware[1]. All of these bugs are now believed to have been fixed. One of the problems is intrinsic to the device (when the receive fifo is cleared in the middle of a cell arriving) and, now fully understood, has been worked-around in the driver software.

Receive performance is always expected to be somewhat lower than the transmit performance on any card. This is due to the inherent additional complexity of checking and verifying headers etc. Also, the EATM card design requires two extra cycles to the card per cell over and above the costs for transmit. These are for reading the status register and writing either the receive CRC data register or the receive CRC continue register.

Once the EATM card had been debugged, the standard configuration programmed by the driver was for it to interrupt if there were more than 16 cells in the FIFO, or if any of the cells in the fifo had the end-of-packet bit set.

5.2.1 Interrupts

When using a high bandwidth polled-I/O interface such as the EATM an enormous interrupt load is placed on the machine and system when receiving data. As a result, receive interrupt processing plays an important part in the overall system receive performance.

Interrupt latency is important, since if it is too high, the fifo buffer memory on the card may overflow. It is also important that the interrupt handler relinquishes the machine sufficiently often that the user process has time to discard the data that has arrived for it and replenish the receive buffer pool.

Another concern is that system clock interrupts are not lost due to the amount of time spent in the EATM interrupt handler. Such events would manifest themselves in the form of a slippage of the time on the system therefore reducing the accuracy of any receive measurements. As described earlier the NT system provides no standard clock synchronisation support, but the complexity of its general interrupt handling gives the impression that this has probably been dealt with correctly.

When an extreme load was placed on the linux system it was discovered that it was in fact losing timer interrupts. Fortunately, having the source code meant that it was possible to investigate why exactly this was occurring. It transpired that although the clock interrupts were not disabled during the EATM driver processing, the EATM interrupts were not in fact disabled during clock interrupt processing! This could lead to the case where an EATM interrupt occurred during a clock interrupt, and the EATM interrupt lasted longer than a clock tick, the tick being lost since recursive clock interrupts were disabled.

Since the linux kernel source code was available, this problem was fixed by ensuring that the clock interrupt was altered to give traditional unix hardclock semantics, where the clock interrupt handler could not be interrupted by any other interrupt routine. It is a testament to the clarity

[1]Although some work had been done with HP-UX, the effort reported here was the first to stress the hardware.

of the linux kernel that, although this change required a changes to the interrupt dispatching assembler and management code, and although the author has little Intel assembler experience, this change worked first time.

5.2.2 Receive Measurements

To measure the receive performance it is necessary to have a traffic source. For these experiments, a real multimedia stream from a ForeSystems AVA200 was used. This stream comprised a 24bit video stream of an entire PAL image using the maximal tile packing factor. Thus the packets being received were 6920 bytes in size (145 cells).

In both systems, a user process existed which was receiving on some VCI. The user process would set up approximately 1 Mbyte of receive buffering (128 buffers of 8 Kbyte each). It would then enter a loop where every time it received a packet it would discard the data and re-queue the buffer for reception.

Another process was run on the machine which printed device driver statistics every second. This allowed the author to detect whether packet loss, or other problems, were occurring, and also the rate in packets per second of reception.

Since the source used in these experiments is a video source, it is not a perfect CBR source; there are inter-frame gaps in the cell stream. This affects the scheduling on the end system: during the video frame it is acceptable for the user process to fall slightly behind in discarding the packets (as long as the buffer is not exhausted), provided that it can catch up during the inter-frame period. This leads to two separate performance numbers in the table 5.2 below, reflecting the maximum peak rate and maximum sustained rate respectively.

	Windows NT	Linux
Sustained	18 Mb/s	25 Mb/s
Peak	22 Mb/s	38 MB/s

Table 5.2: Receive Performance

Chapter 6

Conclusions

This report has described the overall design of the Windows NT ATM device driver. This design has been substantially altered in the course of the project to date due to experience with the NT system.

The requirement for, and description of the DLL for integration of the application interface and signalling component with the device driver has been presented.

This document gives a detailed discussion of the interface, initialisation and operation of the kernel device driver.

The experience derived from the implementation of the NT device driver (and associated test code) when extrapolating to the system in general, gave the impression that NT was not sufficiently light weight or flexible for use as a platform for multimedia terminal equipment research.

To assess this hypothesis, a device driver has also been written for the Linux operating system to be user as a comparison. The design and implementation of this device driver is also described.

ATM bandwidth and performance figures for both NT and Linux on identical hardware are presented. This shows NT to have poor performance compared to Linux, validating the initial suspicion.

The conclusion of this report is that the NT operating system, whilst the first really reliable operating system from Microsoft, and a useful platform for commercial products, is not sufficiently flexible or open for research use in the field of ATM and networked multimedia.

Appendix A

Sample code for NT

This appendix gives some example test code for accessing the device driver from NT. This code is an extract from some of the testing code for the device driver.

```
static int read_stats(HANDLE h, struct eatm_stats *s)
{
    int                 i;
    BOOLEAN             b;

    SetLastError(0);
    if (SetFilePointer(h, 0, NULL, FILE_BEGIN) == 0xffffffff)
    {
        fprintf(stderr, "read_stats: SetFilePointer: error %d for %#x\n",
                GetLastError(), 0);
        return -1;
    }

    SetLastError(0);
    b = ReadFile(h, s, sizeof(*s), &i, NULL);
    if ((!b) || (i != sizeof(*s)))
    {
        fprintf(stderr, "read_stats: error %d at %#x got %d\n",
                GetLastError(), 0x000, i);
        return -1;
    }
    return 0;
}

/* -------------------------------------------------- */

static int tx(HANDLE h)
{
    int i;
    BOOLEAN b;
    unsigned cell[12*2];
    unsigned hdr = TESTVCI;

    for(i=0; i<12+10; i++) cell[i] = i;
    cell[12+10] = htonl(88);

    SetLastError(0);
    if (SetFilePointer(h, 0, &hdr, FILE_BEGIN) == 0xffffffff)
    {
        fprintf(stderr, "write: SetFilePointer: error %d for %#x\n",
                GetLastError(), hdr);
    }

    SetLastError(0);
    b = WriteFile(h, cell, 48+48, &i, NULL);
    if ((!b) || (i!=48+48))
    {
        fprintf(stderr, "WriteFile: error %d at %#x put %d\n",
                GetLastError(), hdr, i);
    }
```

```
    return 0;
}

/* ------------------------------------------------- */

static int rx(HANDLE h)
{
    int i;
    BOOLEAN b;
    OVERLAPPED o;
    unsigned cell[12*2];
    unsigned hdr = TESTVCI;
    DWORD rc;

    memset(&o, 0, sizeof(o));

    o.OffsetHigh = hdr;
    o.Offset = 0;
    if ((o.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)
    {
        fprintf(stderr, "CreateEvent: error %d\n", GetLastError());
    }

    SetLastError(0);
    b = ReadFile(h, cell, 48+48, &i, &o);
    if ((!b) && (GetLastError()==ERROR_IO_PENDING))
    {
        fprintf(stderr, "ReadFile: is pending\n");
        rc = WaitForSingleObject(o.hEvent, INFINITE);
        if (rc != WAIT_OBJECT_0)
        {
            fprintf(stderr, "WaitForSingleObject: error %d\n", rc);
            return -1;
        }
    }
    else if (!b)
    {
        if ((rc = GetLastError()) == ERROR_INVALID_PARAMETER)
        {
            fprintf(stderr, "ReadFile: invalid parameter\n");
        }
        else
        {
            fprintf(stderr, "ReadFile: error %d\n", rc);
        }
        return -1;
    }
    else
    {
        fprintf(stderr, "ReadFile: appeared to work synchronously!\n");
    }

    fprintf(stderr, "got %d bytes\n", i);

    GetOverlappedResult(h, &o, &rc, TRUE);

    fprintf(stderr, "got %d bytes by other method\n", rc);

    for(i=0; i<12*2; i++) printf("cell[%02d] = 0x%08x\n", i, cell[i]);

    return 0;
}

/* ------------------------------------------------- */

static int stats(HANDLE h)
```

```
{
    struct eatm_stats   s;

    if (read_stats(h, &s) < 0)
        return -1;

    printf("ints     = %10u\n", s.es_ints);
    printf("viol     = %10u\n", s.es_viol);
    printf("stray    = %10u\n", s.es_stray);
    printf("full     = %10u\n", s.es_full);
    printf("hdrerr   = %10u\n", s.es_hdrerr);
    printf("times    = %10u\n", s.es_timed);
    printf("nothing  = %10u\n", s.es_nothing);
    printf("\n");
    printf("done     = %10u\n", s.es_done);
    printf("bad      = %10u\n", s.es_bad);
    printf("nobuf    = %10u\n", s.es_nobuf);
    printf("oflow    = %10u\n", s.es_oflow);
    printf("fail     = %10u\n", s.es_fail);
    printf("rxcell   = %10u\n", s.es_rxcell);
    printf("\n");
    printf("cancel   = %10u\n", s.es_cancel);
    printf("rxreq    = %10u\n", s.es_rxreq);
    printf("txreq    = %10u\n", s.es_txreq);
    printf("disables = %10u\n", s.es_disables);
    printf("enables  = %10u\n", s.es_enables);
    printf("\n");
    printf("txcell   = %10u\n", s.es_txcell);

    return 0;
}

/* ------------------------------------------------- */
```

Appendix B

Sample code for Linux

This appendix gives some example test code for accessing the device driver from Linux. This code is an extract from some of the testing code for the device driver.

```
/* -------------------------------------------------- */

static int stats(int f)
{
    struct eatm_stats   s;

    if (ioctl(f, EATMCTL_GSTATS, &s) < 0)
        return -1;

    printf("ints     = %10u\n", s.es_ints);
    printf("viol     = %10u\n", s.es_viol);
    printf("stray    = %10u\n", s.es_stray);
    printf("full     = %10u\n", s.es_full);
    printf("hdrerr   = %10u\n", s.es_hdrerr);
    printf("times    = %10u\n", s.es_timed);
    printf("nothing  = %10u\n", s.es_nothing);
    printf("\n");
    printf("done     = %10u\n", s.es_done);
    printf("bad      = %10u\n", s.es_bad);
    printf("nobuf    = %10u\n", s.es_nobuf);
    printf("oflow    = %10u\n", s.es_oflow);
    printf("fail     = %10u\n", s.es_fail);
    printf("rxcell   = %10u\n", s.es_rxcell);
    printf("\n");
    printf("cancel   = %10u\n", s.es_cancel);
    printf("rxreq    = %10u\n", s.es_rxreq);
    printf("txreq    = %10u\n", s.es_txreq);
    printf("disables = %10u\n", s.es_disables);
    printf("enables  = %10u\n", s.es_enables);
    printf("\n");
    printf("txcell   = %10u\n", s.es_txcell);

    return 0;
}

/* ---------------------------------------------------------------------- */

static int ping(int f, unsigned *at)
{
    int                 i;
    int                 size;
    struct eatm_rbuf    tx[1];
    struct {
        struct eatm_rbufs       hdr;
        struct eatm_rbuf        bufs[3];
    } req;
    fd_set              rdfs;
    struct timeval      t;
    struct eatm_rxinfo  rx;
```

```
    for(i=0; i<12+10; i++) at[i] = i;

    size = 48;

    tx[0].rbr_base = 0;
    tx[0].rbr_len = size - 4;

    at[(size / 4) -2] = htonl((size - 8));

    if (write(f, tx, sizeof(tx)) != sizeof(tx))
        fault("write size=%u", size);

    req.hdr.rb_nelems = 1;
    req.bufs[0].rbr_base = 0;
    req.bufs[0].rbr_len  = size;
#if 0
    req.bufs[1].rbr_base = 4080;
    req.bufs[1].rbr_len  = 24;
#endif /* 0 */

    if (ioctl(f, EATMCTL_SETUPRX, &req) < 0)
        fault("test_prep_rx: ioctl");

    FD_ZERO(&rdfs);
    FD_SET(f, &rdfs);

    t.tv_sec = 10;
    t.tv_usec = 0;

    if (select(f+1, &rdfs, NULL, NULL, &t) < 0)
        fault("test_select: select");

    if (!FD_ISSET(f, &rdfs))
    {
        printf("ping: no response in 10 seconds");
        return -ETIMEDOUT;
    }

    if (read(f, &rx, sizeof(rx)) != sizeof(rx))
        fault("ping: read");

    for(i=0; i<size/4; i++)
        printf("at[%02d]=0x%08x\n", i, at[i]);

    return 0;
}

/* --------------------------------------------------------------------- */

#define TIMES           20480
#define MAXAAL5SIZE     65568

static int blast(int f, unsigned *at)
{
    int                 i, n;
    int                 size;
    struct eatm_rbuf    tx[1];
    struct timeval      t1, t2;

    for(i=0; i<12+10; i++) at[i] = i;

    gettimeofday(&t1, NULL);
    gettimeofday(&t1, NULL);
    do
    {
        gettimeofday(&t2, NULL);
    }
```

```
    while(t1.tv_sec == t2.tv_sec && t1.tv_usec == t2.tv_usec);
    if (t1.tv_usec > t2.tv_usec)
    {
        t2.tv_sec --;
        t2.tv_usec += 1000000;
    }
    printf("difference of %ld.%06lu\n", t2.tv_sec - t1.tv_sec,
           t2.tv_usec - t1.tv_usec);

    for(size = 48 * 2; size <= 9216; size += 48 * 2)
    {
        gettimeofday(&t1, NULL);

        tx[0].rbr_base = 0;
        tx[0].rbr_len = size - 4;

        at[(size / 4) -2] = htonl((size - 8));

        for(n=0; n<TIMES; n++)
        {
            if (write(f, tx, sizeof(tx)) != sizeof(tx))
                fault("write size=%u", size);
        }
        gettimeofday(&t2, NULL);
        if (t1.tv_usec > t2.tv_usec)
        {
            t2.tv_sec --;
            t2.tv_usec += 1000000;
        }
        printf("%5u blocks size %5u took %ld.%06lu\n", TIMES, size,
               t2.tv_sec - t1.tv_sec,
               t2.tv_usec - t1.tv_usec);
    }

    return 0;
}

/* --------------------------------------------------------------------- */

#define NBINS           128
#define SUCKSIZE        8160
#define SUCKBUFSIZE     8192

static int suck(int f)
{
    int         shmid;
    unsigned    *at;
    unsigned    shmsize = NBINS * SUCKBUFSIZE;
    struct {
        struct eatm_rbufs       hdr;
        struct eatm_rbuf        bufs[1];
    } req[NBINS];
    struct eatm_rxinfo          rx;
    fd_set                      rdfs;
    struct timeval              t;
    int                         i;
    unsigned                    n;

    /* Get the memory ready */

    if ((shmid = shmget(IPC_PRIVATE, shmsize, 0600)) < 0)
        fault("shmget");

    if ((at = (void *)shmat(shmid, NULL, 0)) == 0)
        fault("shmat");

    if (shmctl(shmid, IPC_RMID, 0) < 0)
```

```
        fault("shmctl IPC_RMID");

    for(i=0; i<shmsize/4; i+= 1024 / 4)
        at[i] = 0;

    if (ioctl(f, EATMCTL_BIND, shmid) < 0)
        fault("shmbind touched");

    /* Initialise the pending receives */

    for(n=0; n<NBINS; n++)
    {
        req[n].hdr.rb_nelems = 1;
        req[n].bufs[0].rbr_base = SUCKBUFSIZE * n;
        req[n].bufs[0].rbr_len  = SUCKSIZE;

        if (ioctl(f, EATMCTL_SETUPRX, &req[n]))
            fault("suck fullup n=%d", n);
    }

    /* Suck for ever */

    for(n=0; ;n++)
    {
        FD_ZERO(&rdfs);

        for(;;) {
            t.tv_sec = 1;
            t.tv_usec = 0;

            FD_SET(f, &rdfs);

            if (select(f+1, &rdfs, NULL, NULL, &t) < 0)
                fault("select");

            if (FD_ISSET(f, &rdfs))
                break;

            printf("n=%u\n", n);
        }

        if (read(f, &rx, sizeof(rx)) != sizeof(rx))
            fault("read n=%u", n);

        if (rx.rxi_ptr != &req[n % NBINS])
            printf("got unusual block pdu size %u, ctl=%p, n=%u\n",
                   rx.rxi_len, rx.rxi_ptr, n);

        if (rx.rxi_len % 48)
            printf("got unusual length %u\n", rx.rxi_len);

        /* use the same one again */
        if (ioctl(f, EATMCTL_SETUPRX, rx.rxi_ptr) < 0)
            fault("ioctl n=%d, rx.rxi_ptr=%p ", n, rx.rxi_ptr);
    }
}

/* ------------------------------------------------------------ */
```

Bibliography

[**Black94a**] R. Black and S. Crosby. *Experience and Results from the Implementation of an ATM Socket Family.* In USENIX Winter 1994 Conference, pages 143–152, January 1994. (pp 6–2, 6–3, 6–7)

[**Black94b**] R.J. Black. *Explicit Network Scheduling.* Technical Report 361, University of Cambridge Computer Laboratory, December 1994. Ph.D. Dissertation. (pp 6–7, 6–13)

[**Black94c**] R.J. Black, I.M. Leslie, and D.R. McAuley. *Experiences of building an ATM switch for the Local Area.* In Computer Communication Review, volume 24, pages 158–167. ACM SIGCOMM, September 1994. (p 6–15)

[**Dixon91**] M.J. Dixon. *System Support for Multi-Service Traffic.* Technical Report 245, University of Cambridge Computer Laboratory, September 1991. Ph.D. Dissertation. (p 6–7)

[**Hayter94a**] M. Hayter and R. Black. *Fairisle Port Controller Design and Ideas.* In ATM Document Collection 3 (The *Blue* Book), chapter 23. University of Cambridge Computer Laboratory, March 1994. (p 6–15)

[**Hayter94b**] M. Hayter and R. Black. *FPC3 Xilinx Xi5 Design and Notes.* In ATM Document Collection 3 (The *Blue* Book), chapter 25. University of Cambridge Computer Laboratory, March 1994. (p 6–15)

[**Hitatchi91**] Hitachi Semiconductor, 2000 Sierra Point Parkway, Brisbane, CA 94005-1819, U.S.A. *HD64981F EISA Slave Interface Controller (ESIC)*, June 1991. (p 6–12)