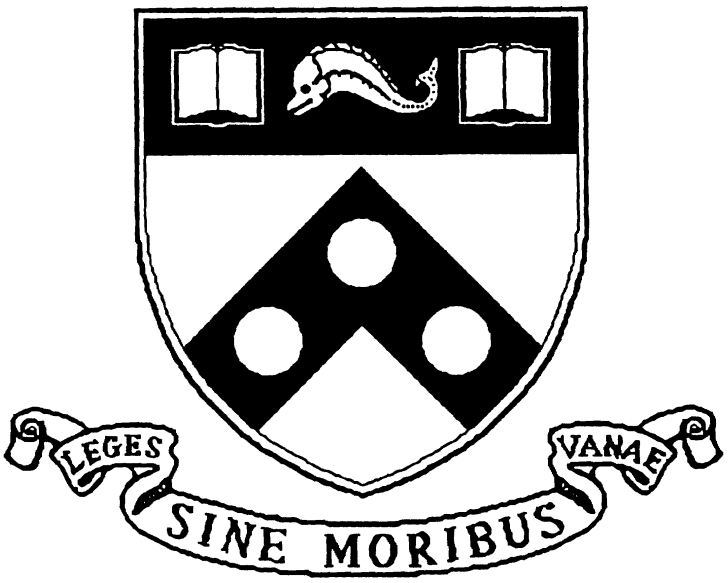
Developing Device Drivers for Character-Class MCA Adapters in AIX Version 3

MS-CIS-93-29
DISTRIBUTED SYSTEMS LAB 27

Michael Massa



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

February 1993

Developing Device Drivers for Character-Class MCA Adapters in AIX Version 3

Michael Massa
Distributed Systems Laboratory
University of Pennsylvania

The purpose of this paper is to introduce the reader to the concepts and techniques required to write a device driver for a character class device on the Micro Channel Architecture (MCA) bus in Version 3.xx of IBM's AIX operating system (denoted by just AIX in this document). It attempts to tie together the major pieces of development information dispersed throughout the AIX manuals. It also includes some information not found in the manuals which is crucial to the development of a working device driver. This document by no means provides complete coverage of the topic, and references for further information will be made liberally throughout. It may best be regarded as a road map to AIX device driver development. The reader is assumed to be proficient in UNIX/AIX applications programming and the C programming language.

Introduction to Device Driver Development

UNIX provides a common interface to all system entities through its file system. An entry in the file system exists for each physical device (eg. console), logical device (eg. pipe), as well as each regular data file. Thus, all system entities may be manipulated by user programs through file system calls (open, close, read, write, etc.). It is the job of the device driver to manipulate a given device to satisfy a request made by a user program through a file system call.

Device drivers are extensions to the AIX kernel, and, as such, operate with the same privileges and restrictions as any other part of the kernel. Kernel code executes under a single process in one of two environments: process and interrupt. In the process environment, code is pageable (unless explicitly pinned into memory), has access to all kernel and user data structures, and may call any kernel subroutine, any kernel extension, a limited number of user-mode system calls, and a limited number of C library functions. Programs executing within the kernel may be preempted only by an interrupt. Code executing in the interrupt environment must be pinned into memory, must not access any data that is not also pinned, may call a small number of kernel subroutines, and must execute within a bounded number of machine cycles. A program executing in the interrupt environment may only be preempted by an interrupt of higher priority. Synchronization between the process and interrupt environments is provided by raising the processor priority in the process environment to disable interrupts.

Device drivers are generally divided into two classes based on the type of device they service: block and character. Block devices are generally random-access storage devices, such as disk drives, which are addressable in fixed-size data blocks. Character devices are generally devices which support serial data streams, such as terminals. These devices may often be viewed simply as a sequentially accessed UNIX data file. By definition, if a device does not fit into the block category, it is a character device. Block and character device drivers differ in the services they provide to the user. Block devices, for example, may contain mountable file systems while character devices may not. This paper will be concerned solely with character devices and the facilities that support them.

Device drivers themselves can generally be broken down into two parts: the device head and the device handler, also known as the top and bottom halves. The device head acts as the interface between the driver and the AIX system-call mechanism or other parts of the kernel. The device head is also responsible for conversion of incoming requests into internally recognizable formats, synchronization within the driver, data buffering, driver state management, accounting, and error recovery. The device handler is responsible for performing the actual device I/O. The device head executes solely within the process environment while the device handler usually executes within the interrupt environment or a combination of both. The device head and handler usually communicate via a set of shared data structures.

It is possible for a single device head to be written to service an entire class of devices. Device-specific handlers are then written to support a given instance of the class. An example of this type of driver would be one for hard disks. The head functions of a hard disk driver are identical for each make of disk, but a unique handler must be written to directly control each individual disk. It is also possible for a single device handler to service many device heads. One example is the AIX SCSI adapter driver. A separate driver head is written for each device attached to the SCSI bus, but a single driver handler services the actual SCSI adapter. Note that these examples, the head and handler are distinct kernel modules. In the former case, the interface is agreed upon internally while, in the latter, the interface format is actually the same as for a device head module. See "Writing Device Drivers for AIX Version 3" (pp. 45-47) for a more detailed discussion of this concept.

The Device Head

The device head acts as the sole interface between the device driver and the AIX system call mechanism and other parts of the kernel. For character devices, the head consists of from six to ten **entry points** (functions) as well as other supporting routines. Most entry points exist to support a file system call (eg. read, write), but some perform functions unique to device drivers.

Major and Minor Numbers

The file system entry for a each device (in the "/dev" directory by convention) contains the **major** and **minor** numbers assigned to the device. The major number uniquely identifies the device within the system. This number is used as an index into the **device switch table**. Each entry in the table contains a set of pointers to the entry points of the driver supporting the device corresponding to the index. In this manner, a file system call on a device special file by a user process is resolved into a call to the appropriate driver function. The minor number is used by each driver entry point to distinguish among logical channels for multiplexed devices (eg. terminal drivers). In AIX, a unique entry may exist in the file system for each major/minor number combination - for example, the /dev/pts/0, /dev/pts/1, etc. entries where the appended number represents the minor number for the device. Another description of device special files, the device switch table and major and minor numbers may be found in Kernel Extensions and Device Support Programming Concepts, Section 2, pp.4-9.

Driver Entry Points

The entry points supported by character device drivers are as follows:

Required Entry Points:

ddconfig	Performs driver and device initialization and termination processing. May handle special driver or device configuration requests.
ddopen	Enables device operation and prepares for data transfer. Allocates internal I/O buffers and enforces policies with respect to how a device may be opened based on the current device state.
ddclose	Reverse of ddopen. Deallocates any internal I/O buffers and updates the device state with respect to how it may be utilized by competing processes.
ddread	Transfers data from a user-supplied buffer to internal driver buffers for eventual transfer to the device. In cases where the head and handler roles are combined, the actual transfer may be performed by this routine as well without the use of internal buffers.

ddwrite — Transfers data from the device, which is stored in internal buffers, to a user-supplied buffer. In cases where the head and handler roles are combined, the actual transfer of data from the device itself may be performed by this routine as well without the use of internal buffers.

ddioctl — Performs device-specific control functions outside of the scope of normal data transfer (eg. flushing output buffers).

Optional Entry Points:

ddmpx — Allocates and deallocates logical channels for multiplexed devices. Called once for each open of a device file before the call to ddopen and once for each close of a device file after ddclose.

ddselect — Checks to see if one or more events have ocurred on the device. At least three events are generally supported:

POLLIN	data is available for reading
POLLOUT	the driver/device is ready to accept more output data
POLLPRI	an exceptional condition has occurred

Select may execute synchronously or asynchronously with respect to the calling process. Asynchronous notification of events is achieved via the selnotify() kernel call.

ddrevoke — Provides support for devices which are in the Trusted Computing Path to a terminal. This function revokes access to a device or channel and is intended to insure that no "Trojan Horses" exist in the Trusted Computing Path.

dddump — Processes dumps of the kernel state to the driver's device during a system crash or debugging. This entry point need not be provided if the device cannot or will not support kernel dumps.

The "dd" prepended to the name of each entry point is replaced in an actual driver by a short identifier representing the specific device. The identifier must make the function name unique within the kernel name space - for example, "ent" represents the driver for the ethernet adapter. A more in-depth description of each of these entry points, their arguments, and their requirements may be found in Calls and Subroutines Reference: Kernel, Section 2, pp. 15-44.

The Device Dependent Structure

The device head entry points act very much like a set of library routines in that they are only executed when a user program makes a system call on a device that they support. Unlike library routines though, the device driver must maintain its state across calls. The repository of most of this state is the **Device Dependent Structure** (DDS). This structure is maintained and accessed by all parts of the device driver. It contains information such as device state flags, device I/O register addresses, pointers to input and output buffers, interrupt timer buffers, and any other information that is needed to manage the device. The part of this structure which contains information, such as the device's bus I/O address, that cannot be known a priori by the driver is passed in to the ddconfig entry point when the driver is initialized. This information is then augmented to include the remainder of the device state information. In some literature, the part of the DDS passed into the driver at initialization time is also called the DDS. In this paper, this portion of the DDS will be called the **Device Dependent Interface** (DDI). This nomenclature is in keeping with the example device driver code supplied by IBM with AIX. In the AIX manuals, however, the DDI

and the DDS are synonymous. A detailed description of the DDI/DDS may be found in Calls and Subroutines Reference: Kernel, Section 2, pp. 10-11.

Synchronization within the Kernel

At any given time, only one process may be executing within the kernel, and it may not be preempted except by an interrupt. If a process needs to wait for an external event to occur, such as the arrival of input data or the expiration of a timer, then the process must be suspended to allow other processing to occur. AIX provides three kernel calls for this purpose: **esleep()/ewakeup()** and **delay()**. The esleep() call suspends the calling process on an **event word**. The process is awakened by another process making the ewakeup() call on the same event word. This combination is often used between a driver head and handler with the head suspending itself until the handler performs some operation and awakens the head. The delay() kernel call suspends the caller for a specified number of timer ticks.

During periods when a driver function is suspended, other processes may attempt to access the same device. Thus, some method of synchronization must be employed within the driver to protect shared entities such as the DDS. To synchronize among concurrently executing kernel functions which share state, AIX provides kernel **locks.** Several types of locks exist. The one most frequently used in device drivers is a word manipulated by the **lockl()** and **unlockl()** kernel calls. The lockl() call blocks until the specified lock is acquired or the request times out. To guarantee mutual exclusion, an entry point must acquire any synchronization locks before performing processing on the shared entities and relinquish them when finished. In most cases, a single lock, acquired at the beginning and relinquished at the end of each entry point, is sufficient to provide the necessary synchronization. Note that the lock must also be relinquished and reacquired when the function suspends itself to wait for an event. A special form of the esleep() kernel call, esleepl(), exists to automatically release and reacquire a specified lock. A more detailed description of kernel locks may be found in Kernel Extensions and Device Support Programming Concepts, Section 1, p. 11.

Accessing User Data

From a device driver's point of view, three distinct groups of virtual address spaces exist in AIX: user address spaces, the kernel address space, and I/O address spaces. A device driver resides and operates within the kernel virtual address space. A user process, and all of its data, resides within its own virtual address space, separate from the rest of the system. Processes within the kernel may access a user address space only through certain kernel calls. Access to the data of the user process on whose behalf a kernel process is operating is relatively simple. Ten calls exist for this purpose: **copyin()/copyout()**, and **copystr()** are similar to the memcpy() and strcpy() library routines. The **fubyte()**, **fuword()**, **subyte()**, and **suword()** services provide the ability to transfer single bytes or words between the kernel and user address spaces. The **uiomove()**, **ureadc()**, and **uwritec()** calls are the ones use most often by device drivers. The uiomove() call transfers a block of data between kernel and user address spaces based on a **uio** structure. One of these structures is passed to the ddread and ddwrite entry points to describe the destination or source buffer for each call. The ureadc() and uwritec() calls move a single character to or from a user buffer described by the uio structure. Note that the terms read and write, as applied to transfers between kernel and user address spaces, are semantically opposite of their normal definitions: ureadc() transfers a character from kernel space to user space and would be used in the ddread entry point to transfer data from the device to the user-supplied buffer. Access to user address spaces other than that of the current user process is provided by the **Cross Memory Kernel Services**. Lower-level virtual memory object manipulation services are provided as well. Access to the I/O address space will be covered in the section on device handlers. See the AIX manual Kernel Extensions and Device Support Programming Concepts, Section 6, pp. 13-18 for an overview of the kernel memory services. See Calls and Subroutines Reference: Kernel, Section 2, pp 12-14 for a description of the uio structure.

I/O Buffering Support

Three different types of buffers are directly supported by AIX. **Character Lists** (Clists) are used to support devices whose I/O consists of relatively slow serial character streams (eg. terminals). Clists are chains of small character blocks. Kernel services to create and manipulate clists are provided. **Memory Buffers** (mbufs) are byte buffers which are slightly larger than clists. Data can be stored directly in each mbuf or in larger, external buffers managed by the mbuf. Mbufs are used primarily to store messages within communications protocols. Kernel services to add and remove header information are provided in addition to other creation and manipulation routines. Block buffers (bufs) serve as the primary data buffers for block device drivers. Bufs are 4096 bytes in size. Services to allocate and manipulate them are part of the kernel. Mbufs and buffers are scarce kernel resources and should not be used to excess. See Calls and Subroutines Reference: Kernel, Section 2, pp. 6-9 for a description of the clist and buf structures. See Kernel Extensions and Device Support Programming Concepts, Section 6, pp.5-6, 8-9 for a listing of buffer services.

Most character drivers will probably require a custom buffering scheme. Memory for buffers is dynamically allocated/deallocated within the kernel address space using the **xmalloc()** and **xmfree()** calls. Memory may be allocated from either pinned or unpinned heaps.

The Device Handler

The device handler is the set of routines which perform physical device I/O. Requests for data transfer will, in general, be made by both the driver head, in response to write or ioctl requests by the user, and by the device itself, in response to incoming data or other external events. Two methods of dealing with these requests can be used: interrupts and polling. The method employed depends on the device hardware. In either case, the handler will be coded in much the same way: as a set of interrupt handler routines with the interrupts generated by either the device itself or by system timers.

Processor Priority & Interrupt Levels

AIX defines the following processor priorities in the header file "sys/m_intr.h":

INTMAX	0	all interrupts disabled
INTEPOW	INTMAX	early power off warning
INTCLASS0	1	device class 0
INTCLIST	INTCLASS0	character list services
INTCLASS1	2	device class 1
INTCLASS2	3	device class 2
INTCLASS3	4	device class 3
INTTIMER	5	timer interrupt
INTOFFL0	6	off-level for class 0 devices
INTOFFL1	7	off-level for class 1 devices
INTOFFL2	8	off-level for class 2 devices
INTOFFL3	9	off-level for class 3 devices
INTIODONE	10	block I/O services
INTPAGER	INTIODONE	page fault
INTBASE	11	everything enabled

The processor normally runs at INTBASE priority. While drivers may specify any interrupt priority for their handlers, MCA device interrupts usually occur at INTCLASS0 - INTCLASS3 priorities while timer interrupts usually occur at INTTIMER priorities. The actual priority is specified in the data structure passed to the interrupt handler registration service. The priority of an interrupt is generally based on the frequency with which it is generated and

the amount of processing necessary to service it. Kernel Extensions and Device Support Programming Concepts (Section 3, p. 10) provides the following guidelines:

Priority	*Service Time (machine cycles)*
INTCLASS0	200
INTCLASS1	400
INTCLASS2	600
INTCLASS3	800
INTOFFL0	1500
INTOFFL1	2500
INTOFFL2	5000
INTOFFL3	5000

The INTOFFL*n* priorities are for **off-level** interrupt processing. They are used when the amount of processing needed to service an interrupt is too large to handle when the interrupt occurs. These interrupts can be scheduled for processing at a later time by the **i_sched()** kernel service. A more detailed description of off-level processing may be found in Kernel Extensions and Device Support Programming Concepts, Section 2, pp. 9-10.

The priority at which an interrupt occurs is not the same as the interrupt level. Each device which is capable of generating an interrupt is assigned a level at which to generate that interrupt. Interrupt levels are also defined in the header file "sys/m_intr.h". Interrupt levels 0-15 are reserved for bus interrupts. Interrupts 16-31 are assigned to the CPU and software. The timer and off-level processing interrupt levels are in the latter range. An interrupt priority is assigned to each interrupt level. Multiple devices may use the same interrupt levels provided they assign the same priority to the shared level. See Kernel Extensions and Device Support Programming Concepts, Section 3, pp. 12-13 and Section 6, pp. 9-10 for a more detailed description of interrupt processing.

Managing Interrupt Handlers

AIX provides seven kernel calls for managing interrupt handlers. A handler is registered for a particular interrupt level with a particular priority by the **i_init()** routine. A pointer to an **intr** structure, defined in the header file "sys/intr.h", is passed as an argument. This structure contains information, such as interrupt level, interrupt priority, and handler address, that is needed to register the handler. The **i_clear()** kernel service removes a handler from use by the system. Four routines exist to synchronize execution between the driver head and handler. The **i_disable()** service raises the priority of the processor to the specified level, disabling all interrupt levels with a lower or equivalent priority. The **i_enable()** service re-enables interrupts at or above the specified level. The **i_mask()** call disables the specified bus interrupt level while the **i_unmask()** service re-enables it. By using these calls, a driver head routine can prevent its handler from executing during a critical section. A more detailed description of interrupts may be found in Kernel Extensions and Device Support Programming Concepts, Section 6, pp. 9-10.

Coding Interrupt Handlers

The handler executes within the interrupt execution environment. The handler and any data it requires (including possibly the DDS) must be pinned into memory before it is registered so that it does not generate page faults. The **pincode()** kernel call performs this service. It takes a function address as an argument and pins the entire object module containing that function into memory. For this reason, the handler code is usually compiled into a separate object module from the head to minimize the amount of pinned code. The handler code may be unpinned using the **unpincode()** kernel service. External data in the kernel address space that is reference by an interrupt handler can be pinned/unpinned into/from memory using the **pin()** and **unpin()** services. Data in a user address space can be pinned/unpinned using the **pinu()** and **unpinu()** calls. The latter facility is needed when performing DMA directly

between user and device buffers. In the interrupt execution environment, a handler is also restricted in the services it may call. Each description of a kernel service in the manual Calls and Subroutines Reference: Kernel indicates from which environments the call may be made. A more detailed discussion of the restrictions placed on interrupt handlers may be found in Kernel Extensions and Device Support Programming, Sections 1 and 4.

When an interrupt handler is registered with the i_init() kernel service, its intr structure is placed at the end of a linked list of intr structures representing all the handlers for its interrupt level. When an interrupt occurs, each handler on the list is called in sequence with a pointer to its intr structure as an argument. The handler is responsible for determining if the interrupt was generated by the device it serves. If it was, the handler should service the interrupt and return the value INTR_SUCC, indicating that it handled the interrupt. If the interrupt was not generated by its device, then the handler should return INTR_FAIL. Thus, the prototype for an interrupt handler function is as follows:

```
int interrupt_handler( struct intr *handler );
```

Using Timers

If I/O requests are to be accepted by polling rather than by hardware interrupts, a timer must be used to periodically gain control of the processor. The **delay()** kernel service can be used to suspend the calling process for a specified number of timer ticks (one tick = 0.01 sec.). The watchdog timer services - **w_clear()**, **w_init()**, **w_start()**, and **w_stop()** - can be used for non-critical events with resolutions of one second or greater. Unlike the delay() service, however, a watchdog timer runs asynchronously with the calling process. When a watchdog timer expires, it generates an interrupt which is handled by the routine specified in the w_init() call. The handler is called in the interrupt execution environment in the same manner as a hardware interrupt handler but is passed a pointer to the structure describing the timer. For critical events with resolutions of at least one nanosecond or which require an absolute expiration time, the **talloc()**, **tfree()**, **tstart()**, and **tstop()** timer routines should be used. This timer behaves much like the watchdog timer. It is described by an **timer request buffer (trb)** containing an **itimerstruct** structure. Note that the **it_value** portion of the itimerstruct is used to hold the timer value. The declarations for these structures are in the header files "sys/timer.h", "time.h", and "types.h". Timer services are listed in Kernel Extensions and Device Support Programming Concepts, Section 6, pp. 23-25.

Device I/O Access

Two methods of device I/O are provided by AIX: Programmed I/O (PIO) and Direct Memory Access (DMA), which includes streaming. Programmed I/O is controlled by the RS/6000 CPU and I/O subsystem while DMA is controlled either by the I/O subsystem or the device itself. When controlled by the device (a DMA master), the data transfers occur asynchronously with other CPU processing. DMA processing is beyond the scope of this paper. A description of DMA may be found in Kernel Extensions and Device Support Programming Concepts, Section 3, pp. 15-17. A description of the DMA services may be found in Section 6, pp. 6-7. An overview of the hardware aspects of DMA may be found in the I/O section of RS/6000 Hardware Technical Reference-General Information.

RS/6000 I/O Architecture

All bus I/O within the RS/6000, whether programmed or DMA is handled by the I/O Channel Controller (IOCC). This component is the interface between the RS/6000 CPU subsystems and any external bus in the system (usually a single MCA bus). The IOCC also has a direct path to memory for DMA purposes. The IOCC consists of a set of control registers and a 64 byte data buffer. The IOCC effectively shields the RS/6000 processor set from all knowledge of the bus architecture. For example, the IOCC automatically converts transfers of half-word or word quantities to/from an 8-bit device register into sequential byte transfers using a process known as **Dynamic Bus Sizing**. The IOCC also supports string movements and multiple word transfers of up to 64 bytes in one CPU

instruction. These services can be used to dramatically increase the throughput of programmed I/O. See the I/O section of the RS/6000 Hardware Technical Reference-General Information for a detailed description of the IOCC and I/O subsystem.

Bus Address Spaces and Programmed I/O

The address space of the Micro Channel bus is split into two parts: bus I/O and bus memory. A device may support either or both address spaces. Bus I/O addresses usually refer to device registers and may be from one to four bytes wide. The total bus I/O address space is 64K bytes wide. The bus memory address space is much larger. Neither of these address spaces is part of the kernel. Processes within the kernel access these address spaces by mapping a portion of them into the kernel's address space using the virtual memory services described in Kernel Extensions and Device Support Programming Concepts, Section 6, pp. 14-15. These services are quite complicated, so macros have been provided in the header files "sys/adspace.h" and "sys/ioacc.h" for I/O operations. A portion of the MCA bus I/O address space can be mapped into the kernel address space using the following macro:

```
caddr_t BUSIO_ATT(ulong bus_id, ulong io_addr)
```

The *bus_id* parameter contains a value which represents the particular bus to be accessed. This value is used to program the control registers of the IOCC and initialize a segment register for use in remapping the address space. The value for the MCA bus is 0x82000030. This value may be hard coded in the driver or passed in through the DDI for flexibility. The *io_addr* parameter is the offset in the bus I/O space at which to begin the mapping. The mapping extends from this point to the end of the bus I/O space. Normally, this parameter is specified as zero to map in the entire I/O bus. The return value of this macro is the address within the kernel address space to which the bus address space is mapped. To construct a pointer to a device register, merely add the bus I/O address of the register to the this pointer (provided *io_addr* = 0).

Once the bus I/O address space has been mapped, it can be accessed via the macros **BUSIO_PUT*x*(p, v)** and **BUSIO_GET*x*(p)** where *x* is L, S, or C for ulong, ushort and uchar, respectively; p is a pointer to the bus address; and v is the value to be transferred. These macros merely perform assignments between two addresses with the appropriate casts. The pointers are cast to **volatile** to prevent the compiler from optimizing the statement in such a way as to corrupt the assignment. This caste is expensive, however. If the pointer to the bus address space is declared to be volatile to begin with, then a simple assignment between the addresses using dereferenced pointers can be used. The bus I/O address space is unmapped from the kernel address space, using the following macro: **BUSIO_DET(caddr_t *bus_p*)** where *bus_p* is the address returned from the BUSIO_ATT call. Note that the bus address space may not remain mapped into the kernel space upon return from the function that created the mapping. This restriction is necessary because the mapping alters the values of the segment registers. Failure to detach the bus address space is very likely to cause the system to crash.

Bus memory access is very similar to bus I/O access and will not be discussed. The IOCC can be custom programmed using macros in the "sys/iocc.h" and "sys/ioacc.h" header files. Once again, see the RS/6000 Hardware Technical Reference-General Information for more information on the IOCC.

MCA Device Initialization

The Micro Channel Architecture provides for software configuration of all MCA adapters using **Programmable Option Select (POS)** registers. This set of eight registers is must be included in all MCA adapters. They are used to provide the adapter with the information necessary for it to communicate with the MCA bus and the IOCC. The first two registers are read-only and contain the 16-bit **Adapter ID**. Each type of MCA adapter has a unique Adapter ID, registered with IBM, which is used during system configuration. The definition of the last six registers, which are read/write, is device-dependent. They usually contain information such as the device's base I/O address, bus interrupt level, DMA level, etc. One exception is the low order bit of POS register 2 (registers are numbered starting with zero). When set, this bit enables the adapter. When not set, the adapter is disabled.

When the system is powered on, each MCA adapter is disabled. POS registers 2-7 must be programmed with initialization values by the system to configure the device. This procedure is similar to the steps required to perform programmed I/O. The following code fragment is typical:

```
caddr_t     bus_p;
caddr_t     posregs_base_p;
uchar       pos_data[6];
int         posreg_num;

   /* attach the IOCC address space to the kernel's address space */
bus_p = IOCC_ATT( IOCC_BID, 0 );

posreg_num = 2;    /* start with POS register #2 */

   /* obtain a pointer to the adapter's POS registers */
posregs_base_p = bus_p + IO_IOCC + POSREG( posreg_num, adap_slotnum );

   /* read in the current contents of POS registers 2-7 */
for (i=0; i<6; i++) {
   pos_data[i] = BUSIO_GETC( posregs_base_p + i);
}

   /*
      modify pos_data in the as needed to configure the adapter
   */

   /* write out the updated data to POS registers 2-7 */
for (i=0; i<6; i++) {
   BUSIO_PUTC( (posregs_base_p + i), pos_data[i] );
}

   /* detach the IOCC from the kernel's address space */
IOCC_DET(bus_p);
```

Again, see the RS/6000 Hardware Technical Reference-General Information and RS/6000 Hardware Technical Reference-Micro Channel Architecture for more details.

Programmed Performance Enhancements

Most of the overhead of programmed I/O is incurred in transferring data from the RS/6000 CPU to the IOCC. This processing requires about 2 microseconds per transfer and is relatively independent of transfer size. The Micro Channel bus cycle is 100 nanoseconds long. Two bus cycles are required to perform a single 32-bit transfer (address + data). Tests showed that an additional 100 nanoseconds per transfer is required for transfers of 32-bit words to 16-bit or 8-bit registers (two 16-bit or four 8-bit transfers). This delay appears to be due to Dynamic Bus Sizing. There is the possibility that the test adapter was configured incorrectly and that the additional delay can be avoided.

For devices with 8-bit registers, Dynamic Bus Sizing provides a method of speeding programmed I/O transfer rates. By specifying a single 32-bit word transfer instead of four 8-bit transfers, the CPU-IOCC overhead is only incurred once. The overhead of dynamic bus sizing is at most 400 nanoseconds, so 5.6 microseconds are saved. In tests performed on an 8-bit device in a lightly loaded system, the best sustained byte transfer rate was 765 Kb/s. The best sustained word transfer rate was 1.7 Mb/s. The test code was similar to the following:

```
temp_p = buf;
for (i=0; i < amount; i++, temp_p++) {
   *io_reg_p = *temp_p;
}
```

Where io_reg_p and buf are pointers to either word or character data.

Further improvements can be made by coding the transfer routines in assembly language. The assembly instructions **stm** and **lm** move up to 16 words (64 bytes) of data between general purpose registers and memory or I/O space in a single instruction. A sustained transfer rate of 2.5 Mb/sec was achieved using this method. Sustained transfer rates in excess of 10 Mb/s are attainable when transferring 32-bit data to 32-bit adapter registers using these assembly instructions.

The Configuration Subsystem

During boot processing in a AIX, all of the devices installed in the system must be identified and initialized. This process, along with general operational management of devices, is performed by the **Device Configuration Subsystem** of the AIX kernel. This subsystem consists of a set of programs and databases which manage information about the devices in the system. Note that in AIX, the term device can mean a physical component in the system or a pseudodevice such as the system console. System devices are organized in a tree-structured hierarchy rooted by the system node. Each node in the hierarchy depends on its parent either physically or logically for its operation. The path defining the ethernet adapter, for example, would be

System -> System Planar -> I/O Planar -> MCA bus -> Ethernet Adapter.

This hierarchy also represents the order in which the system is configured, with each parent being configured before its children in depth-first order.

Configuration information is stored in the databases of the **Object Data Manager** (ODM), an object-oriented database management system which is part of the Device Configuration Subsystem. A general description of every device which could possibly be in the system at any given time is stored in the **Predefined Devices** (PdDv) database. Devices are uniquely identified by a **class**, **subclass**, and a **type** expressed in the form class/subclass/type, reflecting the device hierarchy. An example is the ethernet interface card -adapter/mca/ethernet. This information, along with names of the appropriate device driver and configuration routines, is stored in the PdDv database. Information specific to each individual device, such as bus I/O address, is stored in the **Predefined Attributes** (PdAt) database. The default value, the acceptable range of values, and type information are included for each attribute.

The Define Method

For MCA adapters, the boot configuration process occurs in this manner: The bus is scanned to determine which slots are occupied. If a slot is occupied, the Adapter ID of the adapter in that slot is read. The PdDv database is then searched for the entry corresponding to that Adapter ID. The name of a configuration program known as the **Define Method** is then located within this entry and the program executed. This program copies the information in the PdDv entry for the device to a new entry in the **Customized Devices** (CuDv) database. This database stores information for every device currently configured in the system. Each attribute associated with the device is then copied to the **Customized Attributes** (CuAt) database, and a value (usually the default) is assigned from the range of possible values. A logical name for the device will also be assigned. The device is now considered to be in the **Defined** state. Note that if the device was defined in the system prior to the current system boot, then the information describing it will already be present in the CuDv and CuAt databases and no action is needed. All access to the ODM databases is through the functions in the odm library (lodm) in the /usr/lib directory.

The Configure Method

Once the device has been defined within the system, it must be configured for operation by a **Configure Method**. This program is called after the Define Method in the configuration process for a device (provided the device has

been linked into the device configuration tree via a rule in the **Config_Rules** database). It performs the following major operations:

1) Resolve bus resource conflicts.

2) Load the appropriate device driver module(s) into the kernel.

3) Generate a major/minor number for the device.

4) Create the special file for the device.

4) Build the Device Dependent Structure (DDS).

5) Call the configuration entry point of the driver, passing it the DDS.

6) Obtain any Vital Product Data from the adapter and store it in the CuVPD database.

7) Detect and configure any child devices.

Information needed to perform these operations is obtained by first opening the appropriate ODM databases and locating the relevant entries for the device. Conflicts between bus resources requested by the adapter being configured and the resources already in use by other adapters are resolved by invoking the **busresolve()** system call. The device driver module(s) are loaded into the kernel using one of two system calls: **loadext()** and **sysconfig()**. The loadext() system call is used solely to load, unload, and query the status of a kernel extension module. The sysconfig() call is a general purpose configuration command and can be use to load, unload, and query the status of a kernel module as well as to perform several other functions. The major and minor numbers for a device are generated by calls to the **genmajor()** and **genminor()** system calls. The device special file, based on the device's logical name, is created using the **mknod()** system call.

The driver and device are actually initialized by invoking the sysconfig() system call with the DDS as an argument. The sysconfig() function calls the device driver module at the entry point specified during compilation. Once the driver has configured itself and the device, it returns and the Configure Method continues its processing. When the Configure Method completes successfully, the device is in the **Available** state and may be accessed by user programs. A description of the sysconfig() system call can be found in Section 10 of Calls and Subroutines Reference: Base Operating System Volume Two.

Programs to change the value of a device attribute, unconfigure a device, and undefine a device also exist but will not be covered here. Some devices also support the notion of starting and stopping. Methods to support these state changes may also be written for a given device. Example configuration methods are stored in the "/usr/lpp/bos/samples" directory of most AIX systems. The examples provide the majority of the code needed to develop a working method. It should be noted that many devices use generic methods supplied for the standard system devices. A full description of the structure of each ODM database, the requirements of each method, and useful system calls can be found in Section 4 of Calls and Subroutines Reference: Kernel. A detailed description of the operation of the Configuration Subsystem can be found in Section 7 of Kernel Extensions and Device Support Programming Concepts. A full description of the ODM can be found in the AIX manual General Programming Concepts.

User-level & Run-time Access to the Configuration Subsystem

The configuration subsystem is not part of the AIX kernel and can be accessed by the user at run-time. The **cfgmgr**, **lsdev**, **mkdev**, **chdev**, and **rmdev** commands can be used to perform any remaining configuration processing, list all configured devices, configure a specific device, change the configuration of a currently configured device, and unconfigure a currently configured device, respectively. The **Object Data Manager Editor** (odme) program may

be used to display and modify any of the ODM databases using a screen-editor interface. The command-line programs **odmget**, **odmadd**, and **odmdelete** can also be used to display, add, and delete entries for any ODM database.

Compiling and Linking Device Driver Modules

Each source file of a device driver should be compiled separately into an object file using the -c option of the xlc compiler. Next, all of the object files that will make up a single kernel object module should be linked using the following command:

```
ld -o<module name> -e<module entry point>                          \
   -bimport:<import file> -bexport:<export file>  <etc...>         \
   <object files>                                                  \
   <libraries>
```

For driver head modules, the module entry point will usually be ddconfig. Import files are used to specify the source of external symbols referenced within the object files being compiled. Export files are used to identify the object files being compiled as the source of symbols which will be referenced by other modules. Symbols are uniquely defined within the name space of an executable object module. The name space identifier is the full file path to the executable object module. An example export file for an executable module named "/tmp/foo" with three exported symbols would be:

```
#! /tmp/foo

symbol1
symbol2
symbol3
```

Both variable and function names may be exported. The export file for the module in which the symbols are defined then becomes an import file for the modules that reference them. If symbols in a kernel extension are to be made available to the entire kernel name space, then the name space identified at the top of the export file should be "/unix" since, when loaded, kernel extensions become part of the executing kernel module. If any other name space is specified, then the symbols will only be made available to the modules that explicitly import them. Thus, any symbols shared between driver head and handler modules must be declared in export files and imported accordingly.

The name space identifier is the path to the object module in which the symbols are defined for the following reason: when a file that imports symbols is loaded for execution, the object modules that contain those symbols are also automatically loaded. Thus, a single module can be defined as the device driver in the PdDv and loaded by the Configure Method. As long as this module imports symbols from all of the other modules which comprise the driver, the modules will be loaded together. This strategy is also useful when a single head module supports multiple devices, each with its own handler. The handler module can be specified and loaded, causing the common head module to be loaded as well.

Development Strategy for a First Driver

In developing one's first device driver, some experimentation is in order prior to actually designing and implementing the real driver. The first experiment should be to successfully load a test module into the kernel and verify that it can be accessed. The verification can be accomplished with a test program that calls the module's entry point using the sysconfig() system call. Simply returning an error code should be sufficient evidence that the module did execute. If a debug terminal is attached to serial port #1, then the kernel printf(), an undocumented kernel call which behaves the same as the C library function of the same name but prints its output to the debug terminal, can be used instead. At this point it would be wise to build a set of utility programs to load, unload, and query the status of kernel

modules by name using the loadext() system call (sysconfig requires the kernel module id returned when a module is loaded in order to query or unload it).

Note that there appears to be some mysterious voodoo associated with successfully loading and unloading kernel extensions. If a module has already been loaded and another load request is made, the module will not be reloaded. Instead, a reference count will be incremented (decremented in the case of an unload request). The manuals specify that when the reference count reaches zero, the module is unloaded from the kernel. Often, under conditions not known to the author, a module will not be unloaded when requested. This situation can be maddening when developing a driver since changes are not reflected when an updated module is loaded. Thus, the common practice of always returning an error code of some kind, which changes with each module update, is very useful for catching this condition during development. The system call **slibclean()** can also be used to alleviate this condition somewhat. When called, it unloads all kernel modules whose reference counts have reached zero.

Once a module has been successfully loaded into the kernel and accessed, the Configuration Subsystem should be tackled. Begin by studying the example methods in "/usr/lpp/bos/samples". Next, make a dummy entry in the PdDv database. Then modify the Configure Method to call the previously developed test module and verify that the module is called. Now add an attribute and verify that it can be accessed and passed into the kernel module. Next, add support for major/minor numbers and the device special file. Add multiple entry points to the extension and verify that they can be called once registered in the device switch table. Also verify that the Unconfigure Method can unload the module from the kernel and the Define Method can delete the device special file and relinquish the major/minor numbers. If the driver head and handler are to be two separate modules, verify that two modules can both be loaded and that one can reference symbols within the other.

Now the initial design of the driver should begin. First, develop a state diagram for the driver. Label the transitions with the name of the function that carries out the transition. Don't forget to account for events that must occur when a device is opened and closed (eg. draining of output buffers). Also consider under what conditions a device may be opened or closed, how it may be opened, and by what processes in any given state. The same should be done for each of the other entry points which support file system calls.

Next, design the DDI. Determine what information the driver will need to configure the device and what subset of that information must be provided externally. Once this information is specified, add entries to the ODM for the data and add support to the Configure Method to pass in the proper DDI.

At this point, device I/O should be attempted. Begin construction of the ddconfig entry point by adding code to read in the DDI and write out configuration data to the POS registers. Next, add code to access a register on the device and verify that the data was transferred successfully. Pass the data between the kernel and test program to exercise the user data transfer kernel services.

Once these tasks have been accomplished, it is time to start coding the driver in earnest. A general design suggestion would be to make the driver as generic as possible. Ensure that all of the driver state is encapsulated within the DDS since the driver will probably need to support multiple instances of its device. This task can be accomplished by storing the DDS for each device in an array indexed by device major number (which is unique to each device). This situation demonstrates the importance of the synchronization locks as well. Within reason, design the head to support the broadest class of devices of which the device under consideration is a member. Push device-specific details into the handler or a device-specific head module. The common network device driver head example code located in "/usr/lpp/bos/samples" is an example of this strategy. All drivers for network devices use a common driver head and support an interface that it defines.

References

AIX Calls and Subroutines Reference: Base Operating System Volume 2 for IBM RISC System/6000, IBM Corporation, Section 10.

AIX Calls and Subroutines Reference: Kernel for IBM RISC System/6000, IBM Corporation, Sections 1-2, 4 (pub no. SC23-2198).

AIX Kernel Extensions and Device Support Programming Concepts for IBM RISC System/6000, IBM Corporation, Sections 1-4, 6-7 (pub no. SC23-2207).

Drake, Sam, "Writing Device Drivers for AIX Version 3": AIXpert Journal, Winter 1991, pp. 45-57.

IBM RISC System/6000 POWERstation and POWERserver Hardware Technical Reference-General Infor!¼maCorporation (pub no.

IBM RISC System/6000 POWERstation and POWERserver Hardware Technical Reference-Micro Channel Architecture, IBM Corporation (pub no. SA23-2647).

Exploiting Parallelism in Hardware Implementation of the DES

Albert G. Broscius
Distributed Systems Lab
Dept. of CIS
Univ. of Pennsylvania
Phila PA, 19104-6389 USA
broscius@cis.upenn.edu

Jonathan M. Smith
Distributed Systems Lab
Dept. of CIS
Univ. of Pennsylvania
Phila, PA, 19104-6389 USA
jms@cis.upenn.edu

Abstract

The Data Encryption Standard algorithm has features which may be used to advantage in parallelizing an implementation. The kernel of the algorithm, a single round, may be decomposed into several parallel computations resulting in a structure with minimal delay. These rounds may also be computed in a pipelined parallel structure for operations modes which do not require cryptext feedback. Finally, system I/O may be performed in parallel with the encryption computation for further gain. Although several of these ideas have been discussed before separately, the composite presentation is novel.

1 Introduction

[1] The Data Encryption Standard (DES) is probably the most widely used publicly available secret-key algorithm. Since its introduction by the National Bureau of Standards (NBS) in 1977[FIPS46], DES implementations have improved greatly in encryption rate. Yet, typical computer communication rates have also increased significantly during the same period. Today's high-performance computer networks extend still further the encryption bandwidth needed for adequate performance of secure systems [Giga90]. Thus, we examine means to increase the throughput of a DES implementation to satisfy these demands.

We discuss parallel approaches for several levels of an implementation. At the lowest level, the kernel of the algorithm can be split into several parallel computations for increased speed. By generating subkeys one cycle in advance, the time required can be effectively overlapped with the use of the subkey in the rest of the round operation. An additional overlap can be made of the two stages of exclusive-or (XOR) gates at the expense of increased complexity and gate-count.

One level upward in the hierarchy, the use of multiple round implementations can

[1] This research was supported by NSF and DARPA through the Corporation for National Research Initiatives, and by Bellcore through Project DAWN.

increase computation bandwidth if the DES mode of operation chosen does not require feedback of ciphertext. Of the official modes[FIPS81], this requirement rules out all but the Electronic Code Book (ECB) method. Unfortunately, ECB is known to be susceptible to plaintext frequency-analysis based attacks since multiple identical input blocks result in the same output cryptext block. We discuss in section 3 of this paper a proposed operating method [Feldmeier91] that resists this attack yet does not require feedback of ciphertext.

Finally, at the system level, the processing of I/O concurrent with DES computation provides for continuous operation of the encryption unit. In addition to this buffering, the use of Direct Memory Access (DMA) for encryption allows the host processor to continue other work concurrently with the ongoing encryption.

2 Algorithm Kernel

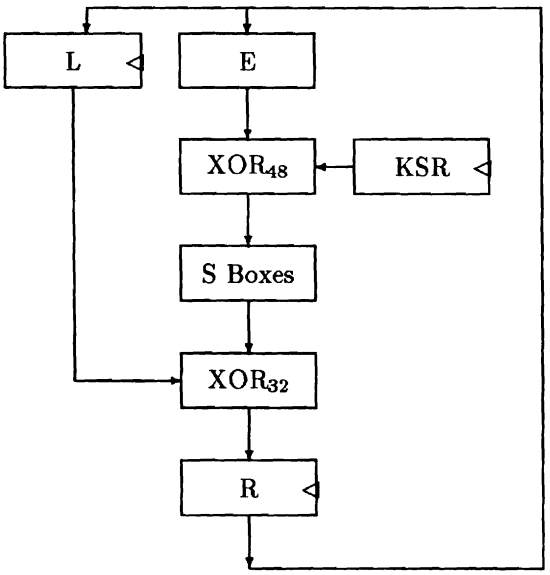


Figure 1: Simple Implementation of Algorithm Kernel

Shift1	Shift2	XOR#1	S Boxes	XOR#2	Reg. Load

Figure 2: Timing Diagram for Circuit of Fig. 1

The kernel of the DES algorithm consists of four operations: key generation, key mixing, substitution table lookup and data mixing between the R and L words. This kernel

is repeated for sixteen iterations with only one key generation parameter dependent on iteration number. This single parameter specifies one or two shifts of the circular registers from which the current key is derived.

A straightforward implementation of the kernel is depicted in Fig.1. The box labeled *KSR* represents the circular shift registers which hold the key data. These are to be clocked either once or twice depending on the iteration number. Once the key has been shifted, the key-mixing box denoted XOR_{48} outputs the modulo 2 sum of the key data with the extended R data after a propagation delay interval. The *S Boxes* then begin their access time delay interval before output of their results. The box marked XOR_{32} then begins mixing in data from the L register. After a propagation delay of an XOR gate, data are ready at the input to the R register. Once a register setup-time has passed, the R and L registers may be clocked once. A register propagation delay later, the cycle may begin once more.

Timing analysis reveals that the critical timing path results from two shifts of the key registers, the keying XOR array, the S Box table lookup, the R-L mixing XOR array, plus the register loading delays. A simplified timing diagram is shown in Fig. 2. The critical path timing defines the limiting rate at which round computation may proceed. Assuming these delays are minimal, the only way to improve the critical path timing is to modify the circuit so that these sequential processes become concurrent. We will now examine several ways to achieve this concurrency.

2.1 Key Parallelism

Separating the key generation from the remaining three stages of the algorithm kernel can reduce the critical path timing. This approach saves delay by updating the key shift register in anticipation of the next iteration simultaneous with the remaining operations in the current iteration. An additional key latch is introduced to buffer the key value for the current iteration on the input to the key-mixing XOR stage as shown in Fig. 3.

This key parallelism was suggested by Diffie and Hellman [Diffie77] in their timing approach for the proposed DES key-search device. They did not include the additional key latch but instead relied on strict control of key shift timing with respect to the overall R-L clock timing to prevent a race condition. Our introduction of the key latch allows greater tolerance in clock provision by ensuring that the key data input to the key-mixing XOR cannot change during the iteration cycle.

Later, a different key-parallel approach incorporating a multiplexer (MUX) was used

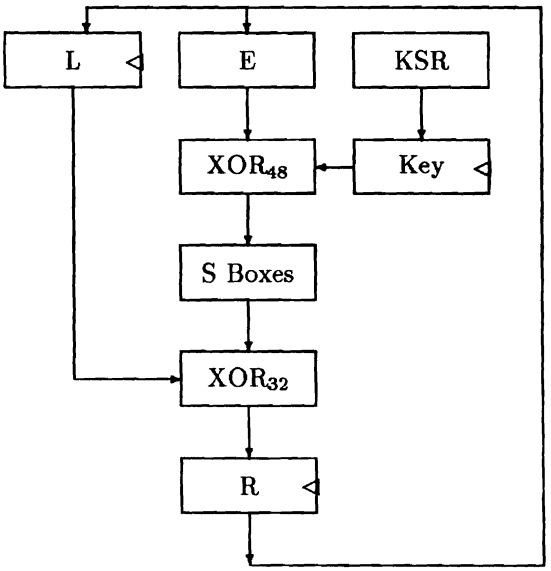


Figure 3: Pipelined Key Generation Algorithm Kernel

	Shift1	Shift2	Key Latch
XOR#1	S Boxes	XOR#2	Reg. Load

Figure 4: Timing Diagram for Circuit of Fig. 3

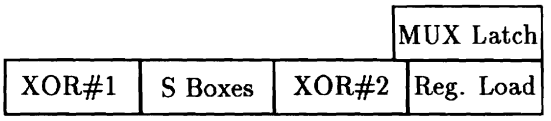


Figure 5: Timing Diagram for Circuit of Fig. 4

by Fairfield et al [Fair84]. Their MUX approach allowed either one or two shifts in either the encryption or decryption direction to be performed in one clocking operation. Additionally, a key loading operation could be selected by the multiplexer. This shortens the time required for the key generation somewhat since the MUX propagation delay is likely to be much lower than a full key shift cycle. More importantly, since the key shift register (KSR) no longer generates intermediate results, as it did when two shifts were required for a given iteration, the extra key latch introduced above to prevent race conditions is no longer necessary so the block diagram reverts to that of Fig.1. A simplified timing diagram for this arrangement is shown in Fig. 5.

2.2 XOR Rearrangement

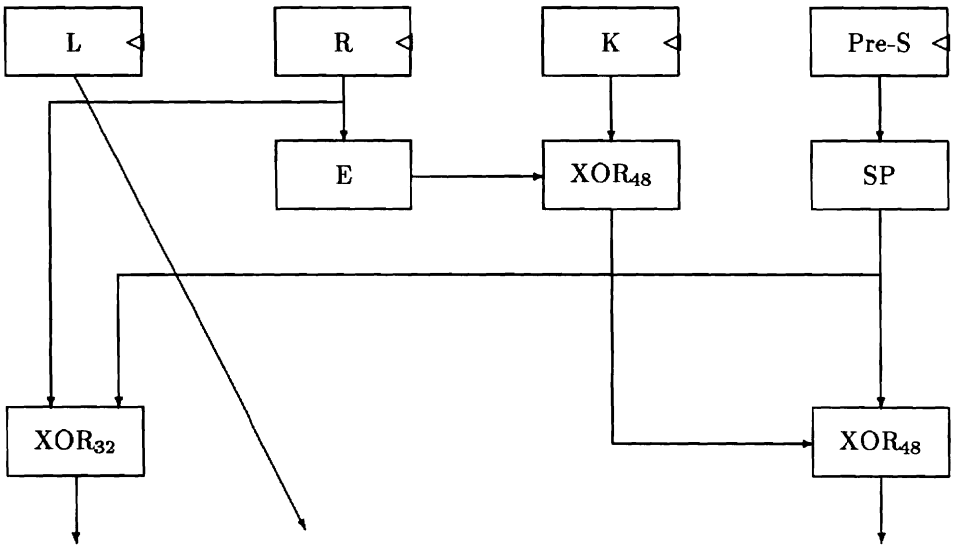


Figure 6: Block Diagram of Datapath with Single XOR Delay in Critical Path

Since the XOR summation is a bitwise linear operation, the order in which XOR operations are performed does not alter their algebraic correctness. Thus, these operations may be grouped (or associated) in any order whatsoever without changing the final results of the combined operations.

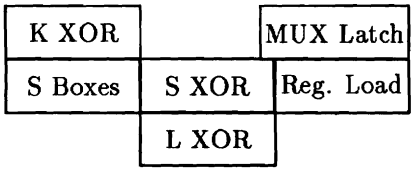


Figure 7: Timing Diagram for Circuit of Fig. 6

If we remove the labeling of R and L in a pair of consecutive rounds of the DES, we observe that there are two stages of XORs, where the second follows the first directly with only the E expansion separating the two. Since E may be commuted with the XOR at the cost of additional bits in the XOR array, we may combine the 48-bit XOR and the 32-bit XOR into a single 48-bit XOR, which would remain in the critical timing path, and a 48-bit XOR which would be computed concurrent with the previous S Box table lookup operation. This transformation also requires the addition of a 32-bit XOR array since the R value is no longer produced in the critical timing path.

3 Multi-Round Parallelism

Using multiple stages in parallel limits the computation of feedback modes of encipherment. Since a parallel implementation begins processing subsequent blocks before completion of a current block's encryption, modes that use the cryptext of a prior block cannot be computed at the full bandwidth that a feedforward mode can achieve. Three of the four modes defined by the NBS for use of the DES require feedback.

A feedforward operation mode proposed by Feldmeier and McAuley appears to overcome the weaknesses of ECB. Their modified ECB mode of operation combines a sequence number with each plaintext block using the XOR operation. This approach should thwart frequency-analysis style attacks since multiple instances of a plaintext block are mapped to different cryptext elements. Using a 64 bit sequence number, cycling of this space would take place in 2^{67} bytes of a data stream. This mode allows independent processing of data elements by avoiding the interdependence of subsequent encryption operations found in feedback modes.

An intermediate alternative between feedback and feedforward modes is the use of multiple interleaved chains. The degree of interleaving can be chosen to allow for as much bandwidth gain through parallelism as needed.

3.1 Pipeline

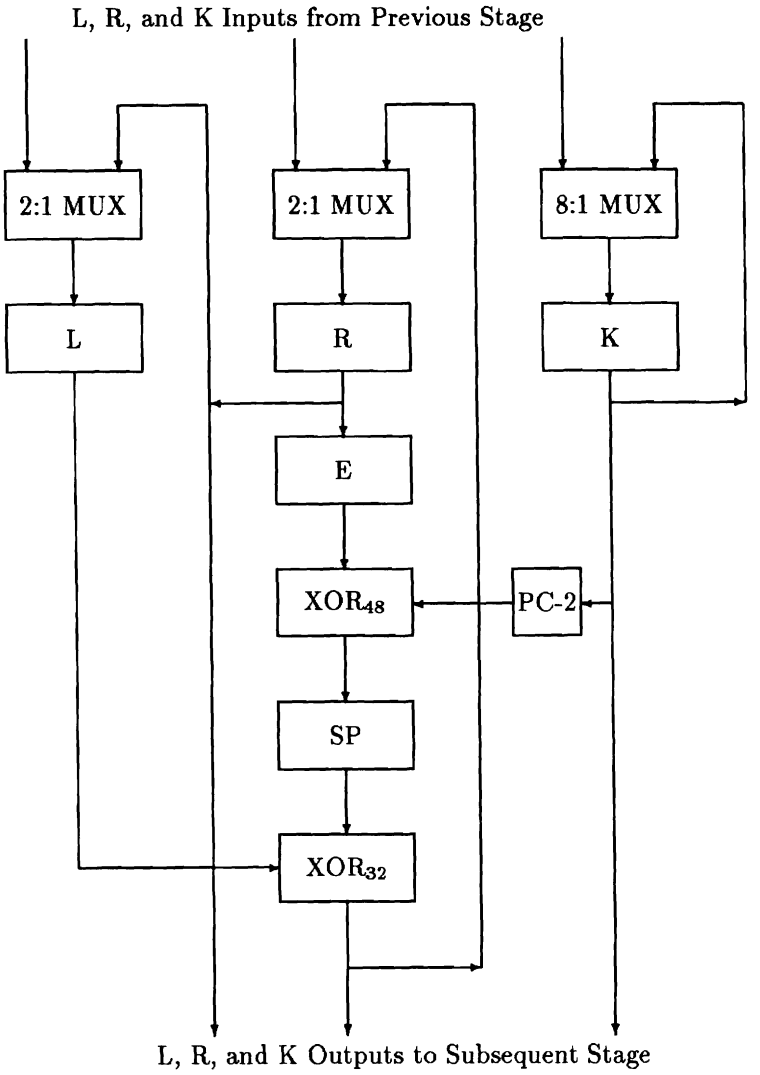


Figure 8: Pipeline Segment for DES with Key Transport

The parallel computation elements may each be configured to implement a fraction of the rounds in a pipeline approach or each element may operate independently in a computation farm approach which we discuss in section 3.2. A pipeline of elements may be configured from two, four, eight or sixteen round implementations. Each element would operate on a 64-bit block for an equal number of cycles needed to partition the algorithm computation. Thus, a two round pipeline would execute eight cycles on each of the processors in the pipeline. Similarly, a four round pipeline would execute four cycles

on each processor.

When keying needs to be updated frequently, the pipeline style allows a matching of the datapath flow with a parallel keypath. In this way, data blocks are accompanied by their key throughout the computation. Switching keys between successive datablocks without flushing the pipeline is made possible since the key and data streams are synchronized. Each stage of the pipeline has the structure depicted in Fig. 8. Note that the 8:1 key MUX actually selects between four different shifted versions of either the current key or the input key from the previous stage.

For infrequent key changes, the tradeoff in keying interconnection may not be worthwhile as compared to maintaining separate key registers for each stage of the pipeline. Each round would then maintain its own key load (shadow) register in this approach. The standard's key shifting sequence would be partially executed on each stage. Since a partial execution of the key schedule would not result in a complete cycling of the keytext, the key would be reloaded from the shadow register when it had completed its share of the computation on a data block.

A limitation of the pipeline approach is the bandwidth ceiling imposed by the number of rounds in the algorithm, sixteen. This means that a single pipeline of processors cannot provide more than sixteen times the encryption bandwidth of a single processor. Additionally, the pipeline suffers in scalability since the number of stages possible is restricted to be a factor of sixteen. This deficit is most notable when considering an upgrade of a pipeline: to gain any increase in bandwidth requires a doubling in computation resources.

3.2 Computation Farm

Instead of a pipeline approach, multiple devices may be configured as a computation farm with each given subsequent plaintext elements to process. This configuration allows for smooth increase in available encryption bandwidth since the number of rounds used need not be a power of two as in the case of the pipeline parallelism.

Managing the farm requires logic similar to that used in FIFO buffers. A counter to keep track of the next available processor and the last busy processor are required. A generalized depiction of the interconnection is shown in Fig.9, using an Input Manager and Output Manager to coordinate operations.

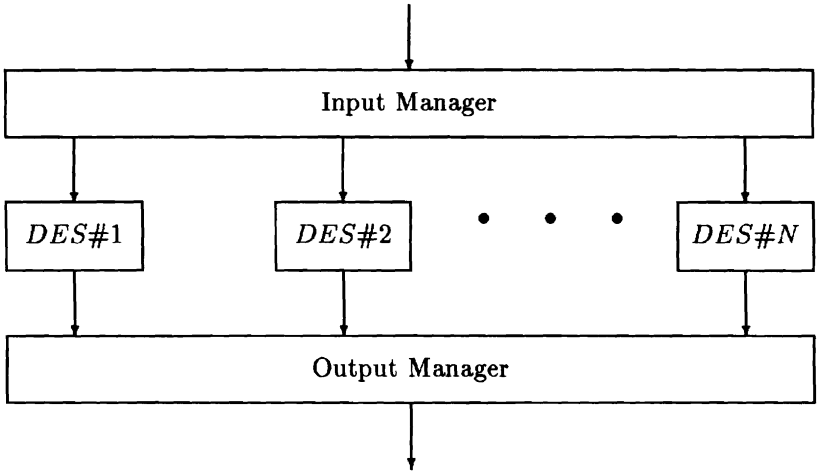


Figure 9: Computation Farm Block Diagram

4 System-Level Parallelism

To maintain constant throughput rates requires careful consideration of the encryption system's interface or input/output section. Overlap of the input, output and encryption processes of subsequent text blocks provides high throughput [Ver88]. Similarly, DMA support decouples the host processor from the encryption function to allow CPU processing of other tasks to proceed in parallel with the encryption request[Anderson87].

5 Conclusion

Parallel aspects of the DES may be exploited at three levels: within the algorithm kernel, through duplication of the algorithm kernel, and in the encryption processor I/O design. Consideration of operation mode also impacts the maximum performance attainable – nonstandard or hybrid operations modes should be studied further as a means of increasing bandwidth without compromising security.

As part of our work with the Aurora network testbed [Giga90], we have developed a DES board [Broscius91] using SSI TTL and MSI PALs using the MUX key register approach. Testing of the wirewrapped prototype indicated an encryption rate of 93 Mbps. Further work on a DMA interface to the MicroChannel interface bus of the IBM RS/6000 is planned. However, recent announcement by VLSI Technology of their VM007 encryption processor [VLSI91] with 192 Mbps performance obsoletes our discrete

approach and will most likely be used in our final version.

References

[Anderson87] David P. Anderson and P. Venkat Rangan, *High-Performance Interface Architectures for Cryptographic Hardware*, EUROCRYPT '87 Proceedings, Springer-Verlag, Amsterdam, 1987

[Broscius91] Albert G. Broscius, *Hardware Analysis and Implementation of the NBS Data Encryption Standard*, MSE Thesis, CIS Dept., Univ. of Penn., May 1991

[Davio83] Marc Davio, Yvo Desmedt, Marc Fosseprez, Rene Govaerts, Jan Hulsbosch, Patrik Neutjens, Philippe Piret, Jean-Jacques Quisquater, Joos Vandewalle and Pascal Wouters, *Analytical Characteristics of the DES* Advances in Cryptology: Proceedings of CRYPTO 83, Plenum Press, New York, 1984

[Denning82] Dorothy Denning, *Cryptography and Data Security* , Addison-Wesley (1982)

[Diffie77] Whitfield Diffie and Martin E. Hellman, *Exhaustive Cryptanalysis of the NBS Data Encryption Standard* , IEEE Computer Vol. 10 No. 6, June 1977 pp. 74-84

[Fair84] R.C.Fairfield, A. Matsuevich and J. Plany, *An LSI Digital Encryption Processor*, Advances in Cryptology: Proceedings of CRYPTO 84, Springer-Verlag, New York, 1985

[FIPS46] National Bureau of Standards, *Federal Information Processing Standard #46: The Data Encryption Standard*

[FIPS81] National Bureau of Standards, *Federal Information Processing Standard #81: Operational Modes of the DES*

[Feldmeier91] Anthony McAuley and David C. Feldmeier, *Minimizing Protocol Ordering Constraints to Improve Performance* , Submitted for publication, Available via anonymous *ftp* from Internet host *thumper.bellcore.com*

[Giga90] Anonymous, *Gigabit Network Testbeds* , IEEE Computer, Vol. 23 No. 9

[Ver88] Ingrid Verbauwhede, Frank Hoornaert, Joos Vandewalle and Hugo de Man, *Security and Performance Optimization of a New DES Data Encryption Chip* , IEEE Journal of Solid State Circuits Vol. 23, No. 3, pp. 647-656, June 1988

[VLSI91] VLSI Technology, Inc., *VM007 Data Encryption Processor* , Tempe, AZ 1991