ALEPH 87-98
DATACQ 87-9
T. Charity
17.11.1987

A Multi-Port Ethernet Driver for OS-9

Tim Charity

EP Division

CERN, Geneva.

14th October, 1987

א

This document describes a device driver for the OS-9/68000 operating system which allows multi-user access to Ethernet/Cheapernet via the LANCE local area network controller. The installation, operation, and maintenance of the driver are described, and the supported configurations are listed. Examples showing how to use the driver from a user process and from within another device driver are given. The driver has been used to provide facilities such as remote login, disk emulation, and inter-networking of OS-9 stations.

This paper has: 30 pages ~~figures annexes~~
If you wish to receive it, please send your request to:
A. Mazzari - EP

Name: _____ Div.: _____

I would like to receive 1 copy of the paper: with annex without annex

Date: _____ Signature: _____

ALEPH 87-98
DATACQ 87-9
T. Charity
17.11.1987

A Multi-Port Ethernet Driver for OS-9

Tim Charity

EP Division

CERN, Geneva.

14th October, 1987

This document describes a device driver for the OS-9/68000 operating system which allows multi-user access to Ethernet/Cheapernet via the LANCE local area network controller. The installation, operation, and maintenance of the driver are described, and the supported configurations are listed. Examples showing how to use the driver from a user process and from within another device driver are given. The driver has been used to provide facilities such as remote login, disk emulation, and inter-networking of OS-9 stations.

Contents

1 Introduction **3**

1.1 Ethernet/Cheapernet 3

1.2 The Am7990 Local Area Network Controller for Ethernet . . 4

1.3 The OS-9/68000 I/O subsystem 4

2 Functionality and Usage **8**

2.1 Principles of Operation 8

2.2 Supported Operations 9

2.2.1 Initialise 10

2.2.2 Open 11

2.2.3 Receive 12

2.2.4 Transmit 13

2.2.5 Restart 14

2.2.6 Cancel Receive 15

2.2.7 Who am I? 16

2.2.8 Associate Circular Buffer and Protocol 17

2.2.9 Disassociate Circular Buffer and Protocol 18

2.2.10 Test Driver 19

2.2.11 Close 20

2.2.12 Terminate 21

2.2.13 Illegal operations 22

2.3 Examples of usage 22

2.3.1 Inside a virtual disk driver 22

2.3.2 Task-to-task communication 24

3 Installation and Maintenance **26**

3.1 Installing the driver 26

3.2 Rebuilding the driver 27

References **28**

1. Introduction

1.1 Ethernet/Cheapernet

Ethernet is a 10 Mbps local area bus network, based on a coaxial cable and a CSMA/CD medium access method[1]. Cheapernet has a lighter and cheaper cable than Ethernet, while providing the same bandwidth and conforming to the same physical protocol specification. It is possible to send messages (frames) to one, several, or all nodes on an Ethernet line simultaneously, depending on the 6-byte Ethernet destination address used. In addition, a 2-byte 'protocol' field in each frame can be used to allow multiple users of Ethernet at a single station. This feature is useful in multi-tasking operating systems, where several different tasks requiring access to the network can be executable simultaneously, and frames are multiplexed to the tasks on the basis of the protocol field. Coordination of access to the hardware responsible for Ethernet/Cheapernet communication is typically performed either by a dedicated task, or via a device driver which is under the control of the operating system. Under VAX/VMS a device driver is supplied with the network interface (DELUA, DEQNA, etc.), and is used by many network applications (such as DECNET) as well as being available to user applications [2]. Under the OS-9/68K operating system it is necessary to write a device driver to suit the particular Ethernet interface in use. This document describes one such driver which has been written for use in the ALEPH data acquisition system, where a large number of 68000-series microprocessors ($\sim$ 100) are linked to a cluster of VAXes via an Ethernet.

1.2 The Am7990 Local Area Network Controller for Ethernet

The Am7990(LANCE) chip is used in many 68000-based microprocessor systems to provide an interface to the Ethernet CSMA/CD local area network. A full description of the chip and the associated Am7991A Serial Interface Adapter can be found in [3]. It performs DMA operations, buffer management, address filtering, error reporting and diagnostics for most aspects of the Ethernet communication. The LANCE is initialised and controlled via a combination of registers and data structures resident within the chip and in memory. A set of four internal registers can be selected and programmed from the host processor, and once enabled the chip accesses memory to obtain further operating parameters from an initialisation data structure. Subsequent transmit and receive operations are managed via two independent ring structures in memory, which are used by the processor and LANCE to synchronise access to data buffers containing the user data. A hardware interrupt can be generated by the chip when a packet is received or transmitted, and when an error occurs during LANCE operation. The register layout is illustrated in Table 1.1, and the data structures which form the memory interface are shown in Table 1.2.

1.3 The OS-9/68000 I/O subsystem

OS-9 is a highly modular operating system originally developed for the Motorola M6809 microprocessor, and subsequently upgraded for the 68000-series microprocessors[4]. All system services (including I/O requests) are processed by the kernel, which can either perform the request itself or delegate it to other system modules, as in the case of I/O. These modules are called File Managers, and generally one file manager is required for each class of device known to the system, such as disks, terminals, pipes, and so on. The purpose of the file manager is to preserve as far as possible the device-independance

of the I/O system calls. Each physical device attached to the system has two modules associated with it: the device driver and the device descriptor. The driver contains the executable code necessary for controlling the hardware, whereas the descriptor is a non-executable table containing configuration parameters. This scheme allows multiple copies of a driver to be instantiated for each device of a given type, by modifying addresses and parameters in the device descriptor. The driver is under the control of one of the file managers, and is called as necessary to perform the requested I/O activities. The most common requests are for simple read and write operations, however more complicated actions, or actions not foreseen by the file manager, may be performed via the wildcard GetStt and SetStt operations. These allow a function code and a list of arguments to be passed to the driver for processing. This technique is used in the multi-port ethernet driver to provide the user with facilities for transmitting and receiving frames as required.

Register	Bit	Name	Description
CSR0	15	ERR	BABL|CERR|MISS|MERR
	14	BABL	Transmit buffer too long (>1519 bytes)
	13	CERR	Collision input not present
	12	MISS	Receiver missed a packet
	11	MERR	Memory error
	10	RINT	Receiver interrupt
	09	TINT	Transmitter interrupt
	08	IDON	Initialisation completed
	07	INTR	BABL|MISS|MERR|RINT|TINT|IDON
	06	INEA	Interrupt enable
	05	RXON	Receiver enabled
	04	TXON	Transmitter enabled
	03	TDMD	Transmit demand before polling
	02	STOP	Stop
	01	STRT	Start
	00	INIT	Initialise
CSR1	15:01	IBBASE	Address of the Init block (0:15)
	00	ZERO	Must be zero
CSR2	15:08	RES	Reserved
	07:00	IBBASE	Address of the Init block (16:23)
CSR3	15:03	RES	Reserved and read as zero
	02	BSWP	Byte swapping during DMA is performed
	01	ACON	ALE assertive state control
	00	BCON	Byte control

Table 1.1: The LANCE Control and Status Registers

Initialisation Block	Address of Transmit Descriptor Ring
	Address of Receive Descriptor Ring
	Ethernet Physical Node Address
	Logical Address Filter
	Mode of Operation
Transmit Descriptor Ring	Transmit Descriptor for 1st Data Buffer
	Transmit Descriptor for 2nd Data Buffer
	Transmit Descriptor for 3rd Data Buffer
	...
	Transmit Descriptor for nth Data Buffer
Receive Descriptor Ring	Receive Descriptor for 1st Data Buffer
	Receive Descriptor for 2nd Data Buffer
	Receive Descriptor for 3rd Data Buffer
	...
	Receive Descriptor for nth Data Buffer
Transmit Data Buffers	Transmit Data Buffer 1
	Transmit Data Buffer 2
	Transmit Data Buffer 3
	...
	Transmit Data Buffer n
Receive Data Buffers	Receive Data Buffer 1
	Receive Data Buffer 2
	Receive Data Buffer 3
	...
	Receive Data Buffer n

Table 1.2: The LANCE↔Processor memory interface

2. Functionality and Usage

2.1 Principles of Operation

The purpose of the driver is to allow users to transmit correctly constructed Ethernet frames to any destination address, and to receive frames from other nodes of the desired protocol type. To transmit a frame the user builds the complete structure in the process-local memory area, and instructs the driver to transmit it. Reception of frames is performed asynchronously; the user notifies the driver of his wish to receive a frame of a given protocol type, and declares a memory location where the incoming frame should be placed. When the driver receives a frame of the correct protocol type, it writes it to the location specified by the user and (optionally) sends a software interrupt or 'signal' to the relevant OS-9 process. This allows the user to continue with other processing or to hibernate pending the arrival of the frame. If no user has requested a frame of the incoming protocol type, the driver will discard the message; buffering of incoming messages is therefore not performed. An exception to this is the case of a remote terminal link, where it is desirable to allow the user on the remote station to 'type ahead' instead of waiting for the echo of the characters typed. For this reason the driver provides a facility for declaring that a protocol type is to be associated with a remote terminal connection. Characters present in incoming Ethernet frames are placed directly into a circular fifo for treatment by a remote terminal driver (see [6]). Broadcast and multicast frames are not filtered and are treated as normal 'point-to-point' frames. Users should use broadcast frames sparingly,

as they must be handled by all stations connected to the network. Note that only frames which are explicitly or implicitly addressed to the station will be received by the driver; the so-called 'promiscuous' mode, where the station can examine all messages on the Ethernet line, is not supported.

The maximum number of simultaneous users is defined by the parameter MAXUSERS in the driver; this affects very slightly the amount of static storage required by the driver, and is currently set to the value 100 which is presumably more than enough for the most demanding requirements.

Note that an imminent release of the driver will provide support for IEEE 802.3 frame format.

2.2 Supported Operations

For each operation in the subsequent list, the relevant OS-9 system service call is shown, together with the arguments which should be supplied in the registers. For more information on the format of OS-9 system service requests consult the technical manual [4].

2.2.1 Initialise

Function	Attach Ethernet to the System	
System call	I$Attach	
Input	d0.b	Access mode (Read_, Write_, Updat_)
	(a0)	Pointer to device name string '/elan'
Output	(a2)	Address of device table entry
Error	cc	Carry bit set in status register
	d1.w	Error code

This causes the Ethernet device driver to be initialised, if this has not already been done. Static storage for the device driver is allocated from the free memory pool at this time; in most configurations this includes allocation of the complete data structure shown in Table 1.2, with the exception of the Transmit data buffers which are not necessary since transmitted frames are taken directly from the process data area. The device driver static storage is cleared, and the LANCE chip is initialised. The interrupt service routine is installed on the OS-9 polling list and interrupts from the LANCE are enabled at initialisation. If the interrupt is generated on the wrong level, then OS-9 will be unable to service it and the system will hang. If the driver has already been attached, the driver initialisation routine is not executed. Typically, the network device is attached after system bootstrap with the shell command 'iniz /elan'. This avoids the overhead of the kernel attaching and detaching the device whenever a user performs an open or close request to the network device.

2.2.2 Open

Function	Open a path to the network	
System call	I$Open	
Input	d0.b	Access mode
	(a0)	Pointer to pathname string '/elan'
Output	d0.w	Path number
	(a0)	Updated past the pathname
Error	cc	Carry bit set
	d1.w	Error code

A path to the network device is opened by the Sequential Character File Manager (SCF). The data structures for the File Manager and the I/O path will be allocated at this time, provided the system has sufficient free memory. A path number is returned to the user which should be used in all subsequent transactions with the network driver. The access mode byte is usually used to specify read or write access; however for the network driver only GetStt operations are necessary, and hence the access mode parameter may be set to zero (no bits set). The only possible pathname string is the name of the device descriptor itself '/elan', and pathnames such as '/elan/input' will cause SCF to return an E$BPNam error. The path number will be the lowest available path number for this process (normally 3 or greater).

2.2.3 Receive

Function	Receive a frame from the network	
System call	I$GetStt	
Input	d0.w	Path number
	d1.w	Receive function code = 1000
	d2.w	Required protocol type
	d3.l	Signal code on reception
	d4.l	Maximum buffer length
	(a0)	Address of receive buffer
	(a1)	Address of buffer length word
Output		none
Error	cc	Carry bit set
	d1.w	Error code

This call notifies the driver that the process is ready to receive a frame from the network with the declared protocol type. The driver records the location and size of the buffer, the address of the length word, and the signal code to deliver when a frame is received. The routine returns to the user process immediately, and it is up to the user to decide if he wishes to sleep pending the arrival of the frame, or to continue with other processing. If the process has no signal handler, then the 'wakeup' signal code (=1) should be specified. A signal code value of zero will cause the process to be aborted on reception of a frame. When a frame is received from the network, the driver interrupt service routine is executed. This checks to see if any process is awaiting frames of the incoming protocol type, and if so it copies the complete frame into the buffer at (a0), up to the maximum specified size (d4). The length of the incoming frame is written to location (a1), and finally the signal code specified in d3 is delivered to the process.

2.2.4 Transmit

Function	Transmit a frame to the network	
System call	I$GetStt	
Input	d0.w	Path number
	d1.w	Transmit function code = 1001
	d2.l	Length of transmit buffer
	(a0)	Address of transmit buffer
Output	d0.l	Return code from transmission
Error	cc	Carry bit set
	d1.w	Error code

The Ethernet frame which begins at (a0) is transmitted by the LANCE onto the Ethernet cable. 'Runt' packets of less than 64 bytes will be extended by taking the first 64 bytes from (a0) onwards. 'Babble' packets of greater than 1514 bytes will be truncated by the driver. The source address field in the frame will be overwritten by the driver with the address used for the initialisation block, usually found from EPROM. The driver modifies a Transmit Descriptor Ring Entry (DRE) to point to the Ethernet frame constructed by the calling process and does not copy the frame to a local buffer. The TDMD bit in CSR0 is set, causing the LANCE to transmit the frame immediately by performing a DMA transfer into the internal silo of the chip. The LANCE makes several attempts to retransmit frames in the event of collisions or other errors. The driver waits for the transmitter interrupt to indicate that transmission is complete before returning to the calling process. If the frame was not succesfully sent a return value of 1 will be placed in d0.

2.2.5 Restart

Function	Re-initialise the LANCE chip	
System call	I$GetStt	
Input	d0.w	Path number
	d1.w	Restart function code = 1002
Output		none
Error	cc	Carry bit set
	d1.w	Error code

This causes the LANCE chip to be re-initialised. The data structures shown in Table 1.2 are reconstructed and the registers are reprogrammed. In the event of a fatal error the device driver will automatically attempt to reinitialise the chip, hence this call should not normally be used unless it is desired to 'force' reinitialisation for test purposes. A better alternative is to Detach and re-Attach the device using the SHELL commands 'deiniz elan' and 'iniz elan'. This will also cause the device driver static storage to be cleared, and all pending receive requests will be cancelled.

2.2.6 Cancel Receive

Function	Cancel a receive frame request	
System call	I$GetStt	
Input	d0.w	Path number
	d1.w	Cancel Receive function code = 1003
	d2.w	Protocol type
Output		none
Error	cc	Carry bit set
	d1.w	Error code

This causes a previously issued Receive request to be cancelled. The protocol type is removed from the driver's list of 'expected' incoming protocols, and any frames received carrying this protocol type will be rejected. A process should not exit with a receive request pending if possible; however the driver does perform checks to ensure that the process has not died, before attempting to send a signal indicating the arrival of a frame. This avoids the possibility of sending a signal to a new process which has subsequently been created with the same process identifier, possibly resulting in untimely death.

2.2.7 Who am I?

Function	Find out the station physical address	
System call	I$GetStt	
Input	d0.w	Path number
	d1.w	Who_am_I function code = 1004
	(a0)	Address of 6-byte buffer
Output		Physical address at (a0)
Error	cc	Carry bit set
	d1.w	Error code

In certain applications the user may wish to find out the Ethernet physical address which is being used by this station. The driver will place a copy of the 6-byte address in the buffer specified at (a0). Normally it is not necessary for the user to know his own Ethernet address since the driver will automatically place it in all outgoing packets from this station. The station address can also be found by inspecting the frame buffers after a transmission request.

2.2.8 Associate Circular Buffer and Protocol

Function	Associate a protocol with a circular buffer	
System call	I$GetStt	
Input	d0.w	Path number
	d1.w	Associate fifo function code = 1005
	d2.w	Protocol type
	d3.l	Size of circular fifo buffer
	(a0)	Address of fifo head pointer
	(a1)	Address of fifo tail pointer
	(a2)	Address of error word
	(a3)	Address of LPRC word in driver static storage
	(a4)	Address of circular fifo buffer
Output		none
Error	cc	Carry bit set
	d1.w	Error code

A circular buffer is declared to the driver for use by a remote terminal driver under OS-9. For full details concerning the remote terminal facility consult the manual [6]. Data contained in any incoming frames of the specified protocol type will be copied into the buffer, and the header pointer will be updated to reflect the presence of more data. If an abort (control-C) or quit (control-E) character is received, the last process to use the terminal port (specified in LPRC) will be signalled appropriately. If the buffer becomes full then incoming data will be discarded. This function is normally executed by the remote terminal driver initialisation routine.

2.2.9 Disassociate Circular Buffer and Protocol

Function	Disassociate a protocol from a circular buffer	
System call	I$GetStt	
Input	d0.w	Path number
	d1.w	Disassociate buffer function code = 1006
	d2.w	Protocol type
Output		none
Error	cc	Carry bit set
	d1.w	Error code

The protocol type specified in d2 is no longer associated with a circular buffer. This function would typically be executed by the remote terminal driver termination routine. Incoming frames with this protocol type will be discarded unless there is a receive request pending.

2.2.10 Test Driver

Function	Test that the driver is installed	
System call	I$GetStt	
Input	d0.w	Path number
	d1.w	Test Driver function code = 999
	(a0)	Input number
Output	d2.l	Contents of (a0)
Error	cc	Carry bit set
	d1.w	Error code

This function copies the contents of (a0) into register d2. It is primarily of use when setting up the driver for a new hardware configuration, to ensure that the driver has been installed and is accessing the user register stack correctly. The ambitious user should inspect the driver source code if he wishes to add extra functionality to the driver in the form of more function codes, in order to perform more sophisticated tests.

2.2.11 Close

Function	Close a path to the network	
System call	I$Close	
Input	d0.w	Path number
Output		none
Error	cc	Carry bit set
	d1.w	Error code

This function is processed by the SCF File Manager, and causes the path specified in d0 to be closed. The user is unable to perform any further transactions with the network driver until a new path is opened with I$Open. When a process dies or exits, all paths which are still opened are automatically closed by the kernel, hence it is normally not essential to use this call but represents good programming practice.

2.2.12 Terminate

Function	Detach the network from the System	
System call	I$Detach	
Input	(a2)	Address of the device table entry
Output		none
Error	cc	Carry bit set
	d1.w	Error code

This causes the network device to be detached from the system, provided it is not still in use by another process (with an open path to the device). The device driver and descriptor modules may be lost from the system if their link count has fallen to zero, or if they were not present at system bootstrap. The device driver stops the LANCE chip and removes the interrupt service routine from the IRQ polling list maintained by the kernel. Static storage is returned to the free memory pool.

2.2.13 Illegal operations

The principal illegal operations associated with the network device driver are I$Read and I$Write. They do not cause any output or input to or from the network but they will not return an error. All operations other than initialisation and termination are handled using the I$GetStt (or I$SetStt) system calls with the appropriate function codes. Function codes unknown to the driver will cause an error of E$UnkSvc to be returned to the user. Also note that although the driver resides under the SCF file manager, it is not a standard SCF device. Consequently output may not simply be redirected to the network with the SHELL redirection operator. The driver is intended as a low-level interface to the network for use directly by user tasks, or by other device drivers providing more facilities.

2.3 Examples of usage

2.3.1 Inside a virtual disk driver

One of the facilities which has made use of the multi-port Ethernet driver is a network disk system, whereby OS-9 files may be stored on virtual disks physically resident on a VAX connected to Ethernet [7]. Inside the driver's initialisation routine a path to the Ethernet driver is opened:

```
/* Open the network path */

        epath = open("/elan",1);
        if( epath==-1 )return 246;
```

Whenever a disk sector is read or written, the disk driver calls the Ethernet drivers to transfer sectors of data and acknowledgements of the transfers:

```
      {
      int len=0,signal=1,protocol=0x8008;
      /* First send the request */
      if( eth_write( epath,&request,RQ_SIZE )!=0 {
      /* Now issue the receive request */
         eth_read(epath,&reply,protocol,signal,&len,RP_SIZE);
      /* Wait for the reply signal */
         {
#asm movem.l d0/d1,-(a7)
       move.l #1000,d0
       os9 F$Sleep
       movem.l (a7)+,d0/d1
#endasm
         }
        if( len!=0 ){ /* Reply received */
        ...etc
```

The structures `request` and `reply` have already been constructed with the correct Ethernet header information, and the data has been placed in the data area. The structure definition is:

```
      typedef struct net_request {
       /* Ethernet information */
       uchar dest[6],source[6],protocol[2];
       /* Application data */
       uchar request_data[1500];
      } request_t;
```

The functions `eth_write` and `eth_read` provide a convenient interface to the I$GetStt functions from 'C'. If you have received your copy of the network

driver via a standard route you should also have these routines, which are also present in the library file 'elib.l':

```
eth_write: movem.l d1-d2/a0,-(a7)
        move.l 16(a7),d2
        movea.l d1,a0
        move.l #1001,d1
        os9  I$GetStt
        movem.l (a7)+,d1-d2/a0
        rts
eth_read: movem.l d1-d4/a0-a1,-(a7)
        move.l 28(a7),d2
        move.l 32(a7),d3
        movea.l 36(a7),a1
        move.l 40(a7),d4
        movea.l d1,a0
        move.l #1000,d1
        os9 I$GetStt
        movem.l (a7)+,d1-d4/a0-a1
        rts
```

2.3.2 Task-to-task communication

It is possible for processes to communicate with each other by using raw Ethernet frames, sent and received by a multiplexing driver. This can be used for communication between a VAX process and an OS-9 process, as in the following example 'bouncer' program:

```
        sigh( val )
        int val;
        { if( val==2 )exit();
```

```
}
main() /* Idiotic bouncer program */
{ int left,path,ret,len;
 short protocol=0x600d;
 unsigned char buf[1600];
/* Install the signal handler */
intercept( sigh );
path = open( "/elan", 1 );
if( path==-1 )exit(errno);
for(;;){ /* Bounce packets until killed */
 ret = eth_read( path,buf,protocol,999,&len );
 left = sleep(3); /* Hibernate until a packet comes */
 if( left ){
 printf("Bouncing %d bytes\n",len );
 ret = eth_write( path_,buf,len );
 printf("Return from write: %d\n",ret );
 } else {
 printf("Nothing received\n");
 }
 }
}
```

This program also makes use of the functions `eth_read` and `eth_write`. Note that it is the responsibility of the calling process to sleep pending reception of a frame.

3. Installation and Maintenance

3.1 Installing the driver

To install the driver it is necessary to load the modules *elan* and *q0driv* from the distribution disk, if they are not already in EPROM. The driver is initialised by the SHELL command 'iniz elan'. If applications using the driver intend to sleep when awaiting packets, the clock should be running. To deinitialise the driver and stop the LANCE when no processes have paths open to the device, issue the SHELL command 'deiniz elan'. Under normal conditions this will never be necessary, and the driver can be left permanently installed. The exception is when a new version of the driver is being developed, in which case it is necessary to deinitialise the driver before the new version is installed. Frequent initialisation and deinitialisation will lead to memory fragmentation due to the driver static storage being repeatedly allocated and deallocated from the free memory pool. The installation steps are summarised below.

- Reboot the system
- Start the system clock with 'setime'
- Load 'elan' (if not in EPROM)
- Load 'q0driv' (if not in EPROM)
- Iniz elan

3.2 Rebuilding the driver

The files necessary to reassemble, recompile, and link the driver are listed in Table 3.1.

elanc.c	'C'-source code of the driver
elanc.h	'C'-header file used in elanc.c
elana.a	Assembler interface to elanc.c
elan.a	Source file for the descriptor
elanlin	SHELL procedure file
oskdefs	OS-9 constant definitions
sys.l	OS-9 symbol definitions
math.l	OS-9 symbol definitions

Table 3.1: Component files for the multi-port Ethernet driver

Examine these files carefully before contemplating any modifications. The bulk of the driver is written in 'C', and the file 'elana.a' provides a small assembler interface to the system by pushing registers onto the stack consistent with the OS-9 'C' calling convention. For more information on how to write OS-9 device drivers in 'C', consult [8]. The only areas where the driver code may need to be changed prior to installation are indicated by `#ifdef ...#endif` clauses in 'elanc.c'. It may also be necessary to change the installation of the interrupt service routine if the LANCE is not on the level 4 autovector. This is performed in 'elana.a'. This will be necessary if your system does not correspond to one of the supported configurations listed in Table 3.2.

To rebuild the driver and descriptor, copy all the files to the ramdisk or execution directory. Execute the SHELL procedure file 'elanlin' which will perform the assembly, compilation, and linking. This will need to be modified if the assembler/compiler/linker are not in the module directory

Option	Description
EB	Aleph Event Builder
TPP	Aleph TPC Processor
LRT	LRT Filtabye 25.0 VME board
VIP	Bonn VIP 68010 VME board
E3	Eltec EUROCOM-3 system

Table 3.2: Supported configurations for the LANCE Ethernet driver

or execution directory. If the system in use is 68020-based, it is preferable to use the c68020 compiler and r68020 assembler by modifying the relevant lines in 'elanlin'. It is necessary to modify the compiler command line to select the driver configuration desired. The '-dXXX' option specifies which of the `ifdef` clauses should be selected, where XXX is one of the options listed in Table 3.2. With a slight modification the driver has also been used in a multi-processor environment, with several processors sharing one LRT board in a VME crate.

References

[1] *Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications*, DEC, Intel, and Xerox corporations.

[2] *System Programming*, VAX/VMS Volume 10a, Digital Equipment Corporation, USA (1986).

[3] *The Am7990 Local Area Network Controller for Ethernet*, Advanced Micro Devices (1983).

[4] *OS-9/68000 Operating System Technical Manual*, Microware Systems Corporation, Des Moines, USA (1987).

[5] *Filtabyte 25.0 Programming Manual*, L.R.T. Limited, Reading, England (1985).

[6] *An OS-9 Remote Login Facility over Ethernet*, T.Charity, Aleph Online Note (1987) (in preparation).

[7] *An OS-9 Network Disk Facility over Ethernet*, T.Charity, Aleph Online Note (1987) (in preparation).

[8] *Writing OS-9 Device Drivers in 'C'*, T.Charity, Aleph Online Note (1987).