inmos®

IMS B431 Ethernet TRAM

Incorporating the IMS F006A support software.

SGS-THOMSON MICROELECTRONICS

INMOS is a member of the SGS–THOMSON Microelectronics Group

72 TRN 235 00

December 1991

72 TRN 235 00 December 1991

Contents

1 Introduction **1**

1.1 Document structure 1

1.1.1 Conventions 1

1.2 Background 1

1.2.1 Prerequisites 2

2 Installation **3**

2.1 IMS F006A software 3

2.2 IMS B431 Ethernet TRAM 3

2.3 Connecting to Ethernet (10BASE5) 4

2.3.1 AUI connection 5

2.3.2 AUI power 6

3 IMS F006A overview **7**

3.1 Components 7

3.2 IMS B431 device driver 7

3.3 IMS F006A interface libraries 8

3.4 Example programs and source code 9

3.5 Environments 9

3.5.1 Development Environment 10

3.5.2 Target System Environment 10

4 IMS F006A libraries **13**

4.1 Interface procedures 13

4.1.1 B431_Init_Normal() 14

4.1.2 B431_Init_Loopback() 16

4.1.3 B431_Start_Ether() 18

4.1.4 B431_Tx_Packet1() 19

4.1.5 B431_Tx_Packet2() 20

4.1.6 B431_Reset_Stats() 21

4.1.7 B431_Stop_Ether() 21

4.1.8 B431_Terminate() 22

4.1.9 B431_Ether_Stats() 22

4.1.10 B431_Waitfor_Event() 23

4.2 Diagnostic procedures 26

4.2.1 B431_Internal_Loopback() 26

4.2.2 B431_External_Loopback() 28

4.3 IMS B431 Device Driver 29

4.3.1 Debugging support 30

4.4 Using B431_Load_Driver() 33

5 **IEEE 802.3 CSMA/CD Ethernets** 35

5.1 IEEE 802.3 CSMA/CD Ethernets 35

5.1.1 Packet structure 36
5.1.2 CRC algorithm 37
5.1.3 Addressing 38
5.1.4 Retry algorithm 38
5.1.5 Ethernet statistics 39
5.1.6 Time domain reflectometer 40
5.1.7 Heartbeat monitor 41
5.1.8 Performance 41

6 **Detailed hardware description** 43

6.1 Data structures 43

6.1.1 Buffer and descriptor ownership 45
6.1.2 Data chaining 45

6.2 Software structure 45

6.3 Initialising 47

6.3.1 The initialisation block 47
6.3.2 CSR0–CSR3 49
6.3.3 Summary 53

6.4 Receiving 54

6.4.1 The receive message descriptor 54

6.5 Size and number of receive buffers 57

6.6 LANCE actions during packet reception 57

6.7 Receive driver actions 58

6.7.1 ERR is 0 58
6.7.2 ERR is 1 59
6.7.3 ENP is set (and ERR is clear) 59

6.8 Transmitting 60

6.9 The Transmit message descriptor 60

6.9.1 Transmit message descriptor 0 (TMD0) 60
6.9.2 Transmit message descriptor 1 (TMD1) 61
6.9.3 Transmit message descriptor 2 (TMD2) 61
6.9.4 Transmit message descriptor 3 (TMD3) 62

6.10 LANCE actions during transmission 63

6.10.1 Failure to transmit 63

6.11 A typical transmit driver 64

6.11.1 BUFF and ERR are clear 65
6.11.2 BUFF is set 65
6.11.3 ERR is set 65

6.12 Interrupts, errors and error handling 68
6.12.1 Interrupt handling 68
6.12.2 Errors 68
6.13 Self Testing 70
6.13.1 The MODE register 70
6.13.2 Loopback tests 71
6.14 IMS B431 TRAM engineering data 73
6.14.1 Connectors and pin allocations 73
6.14.2 Pin descriptions 73
6.14.3 Memory Map 76
6.14.4 Mechanical details 76

7 References 77

Appendices 79

A Directory structure 81

B Example programs 83

B.1 IMS F006A example programs 83
B.1.1 C Example 83
B.1.2 occam Example 87

1 Introduction

1.1 Document structure

This document is intended as a systems developers guide to using IMS F006A with the IMS B431 Ethernet TRAM. It is split into the following chapters.

Chapter 2 describes software and hardware installations.

Chapters 3 and 4 describe the software development environment and facilities provided by the IMS F006A. Use of the IMS B431 device driver in conjunction with the procedural interface libraries is discussed, and for each library procedure a full specification is given. ANSI C and occam programmers should find that all the information required to produce application software is contained entirely within this part of the manual.

Chapter 5 provides background information on the IEEE 802.3 CSMA/CD networking standard (Ethernet). It describes packet format, addressing and the logical address and CRC algorithms. It also specifies the exact meaning and interpretation of the Ethernet statistics gathered by IMS F006A. Details of expected Ethernet performance levels are also provided.

Chapter 6 describes the IMS B431 Ethernet TRAM hardware in detail, it describes the Ethernet interface chip and how it is programmed. This section will only be of interest to programmers who wish to write their own device driver for the IMS B431 TRAM - where special requirements preclude the use of the IMS F006A software.

1.1.1 Conventions

Throughout this manual reference to software routines and constants provided in C and occam will be made using ANSI C syntax. Equivalent occam names may be derived by substituting occurrences of the '_' (underscore) character with a '.' (period) character as appropriate.

1.2 Background

The IMS F006A is a software support package for the IMS B431 Ethernet TRAM. It is intended for those developers wanting to construct transputer systems incorporating IEEE 802.3 Ethernet attachment, but who are not interested in the low level programming details of the Ethernet interface hardware.

The IMS F006A software consists of a device driver for the IMS B431 Ethernet TRAM and procedural interface libraries for ANSI C and occam which access the device driver via a transputer channel pair. This facilitates the creation of transputer programs to establish and engage in packet level communication on IEEE 802.3 Ethernet based local area networks (LANs).

IMS F006A is compatible with INMOS software development toolsets. Systems developers incorporate IMS F006A with their own application software using an appropriately selected toolset.

1.2.1 Prerequisites

In order to develop with IMS F006A the following (minimum) environment is required:–

Hardware

- IBM PC/AT or compatible personal computer
- IMS B008 IBM PC/AT TRAM Motherboard
- IMS B431 Ethernet TRAM (Size 2)
- A compute TRAM such as the IMS B404 (Size 2)

Software

- IMS D7214 ANSI C Toolset for IBM PC/AT

or

- IMS D7205 occam 2 Toolset for IBM PC/AT

2 Installation

2.1 IMS F006A software

The installation of IMS F006A requires at least 800 kBytes of free disk space be available on the host computer system hard disk.

To install IMS F006A from floppy disk drive `A:` onto hard disk drive `C:` of an IBM PC/AT or compatible computer, proceed as follows:–

1 Insert the floppy disk into floppy disk drive `A:`.

2 Change the current working directory to `C:\`.

3 At the operating system command line, type `a:install a c`.

4 Respond as appropriate to prompts made by the install program.

2.2 IMS B431 Ethernet TRAM

Since the IMS B431 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B431 may be supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

Plug the IMS B431 into the motherboard. Where the IMS B431 is being used with an INMOS motherboard, the yellow triangle marking pin 1 on the IMS B431 (see Figure 2.1) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot.

Should it be necessary to unplug the IMS B431, it is advised that it is gently levered out while keeping it as flat as possible. As soon as the IMS B431 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

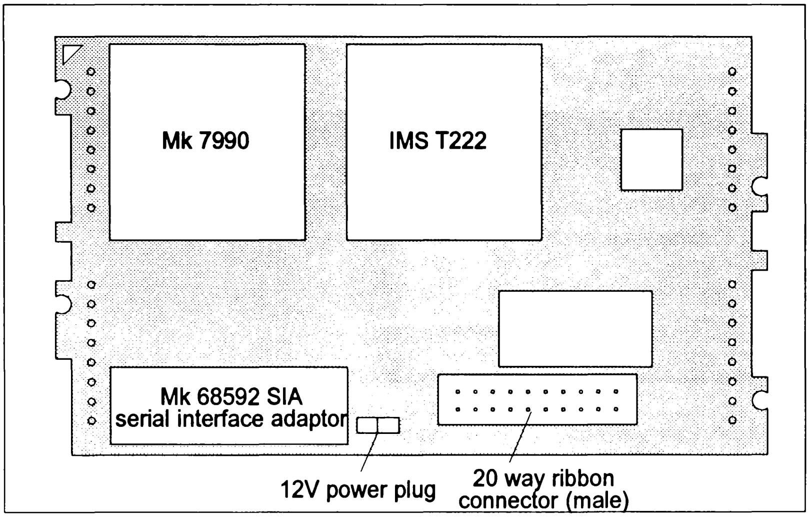
Mk 7990
IMS T222
Mk 68592 SIA serial interface adaptor
12V power plug
20 way ribbon connector (male)

Figure 2.1 IMS B431 board layout

2.3 Connecting to Ethernet (10BASE5)

In an Ethernet (10BASE5) system, nodes are connected to the Ethernet coax by means of an *media access unit* (MAU) and an *attachment unit interface* (AUI) cable. The MAU is a specially designed housing incorporating a transceiver device. The MAU is clamped to the Ethernet coax; it penetrates the coax making contact with the signal conductor without interrupting traffic on the LAN. The node is connected to the MAU transceiver by means of an AUI cable. Ethernet supports a maximum cable length (without repeaters) of 500 metres.

When connecting the IMS B431 to an Ethernet system, the AUI connector on the IMS B431 is used to connect to a separate MAU. Figure 2.2 shows how the IMS B431 should be connected to an Ethernet system. It is intended that the 15-way D-type connector should be mounted in a suitable bulkhead or front panel on the equipment into which the IMS B431 is installed.

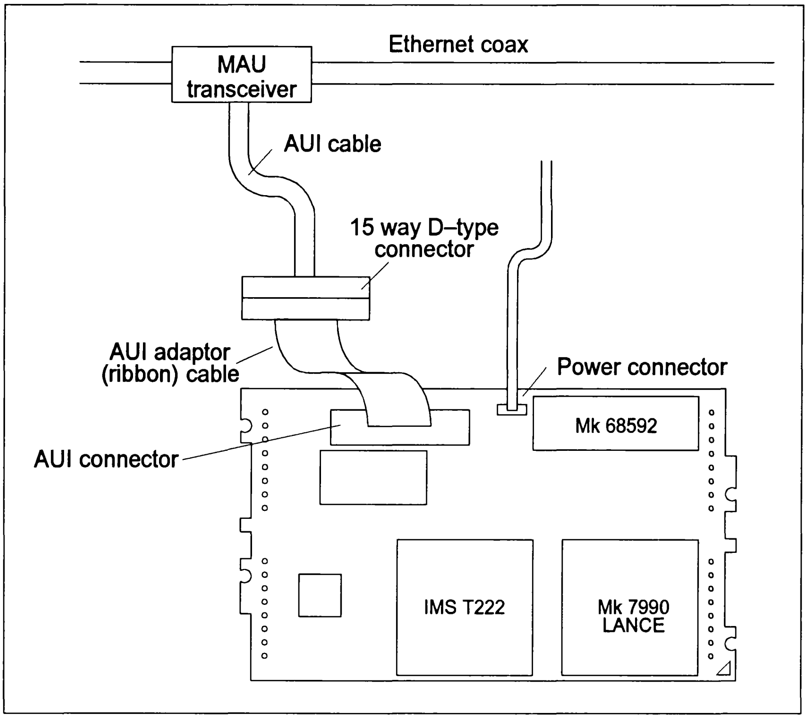

Figure 2.2 Connecting the IMS B431 to an Ethernet system

2.3.1 AUI connection

To reduce the overall height of the IMS B431, a 16-way male header is used instead of the standard 15-way D-type; the pinout of this connector is defined in table 2.1. A short length of adaptor cable is supplied for connection to a standard AUI cable. AUI cables should have a shielded, twisted pair for each signal or power pair; each pair should have a characteristic impedance of 78 ± 5Ω.

The RX ±, TX ± and COLL ± signals on the AUI connector are transformer isolated.

Name	Function	Pin No.
COLL+ COLL–	collision pair	4 3
TX+ TX–	transmit pair	6 5
RX+ RX–	receive pair	10 9
+12V 0V	power pair	11 12
GND	shield	1,2

Table 2.1 AUI connector pinout

2.3.2 AUI power

The Ethernet specification requires that the AUI cable must supply +12V at 0.5A to the MAU. Since there is no +12V supply on the IMS B431, this must come from an external power source. A +12V power source capable of supplying 0.5A should be connected to the power connector on the IMS B431. The IMS B431 routes power from this connector to the AUI connector. The pinout of the power connector is shown in table 2.2.

Name	Function	Pin No.
+12V In 0V In	AUI power pair	1 2

Table 2.2 Power connector pinout

Pins 1 and 2 connect directly to the AUI connector power pair pins.

3 IMS F006A overview

3.1 Components

IMS F006A consists of the following components:–

- IMS B431 device driver
- ANSI C and occam interface libraries
- Include files
- Example programs and configuration files
- Source code of interface libraries

3.2 IMS B431 device driver

The IMS B431 device driver runs on the IMS B431 Ethernet TRAM and is responsible for:–

1 Controlling the Ethernet interface hardware.

2 Performing diagnostic tests upon the Ethernet interface hardware and buffer memory areas.

3 Collecting and making available statistics concerning the operation and performance of the Ethernet interface.

The device driver's primary role is to provide a buffered interface for transmitting and receiving IEEE 802.3 structured Ethernet packets. Packets sent to the device driver for transmission are buffered in a first in first out (fifo) queue where they wait to be transmitted. Similarly, packets received by the Ethernet interface are queued until read. Two transputer channels connect the device driver to application software running on an adjacent transputer. Packets are sent to the device driver by calling a procedure from the IMS F006A interface library with the channel to the device driver passed in as a parameter. In the other direction, packets are received by calling a different procedure to wait for their arrival.

Typically, application software will fork into two parallel processes to handle this concurrent activity. The arrangement is shown in the following diagram, one process is responsible for transmitting packets while the other handles their reception.

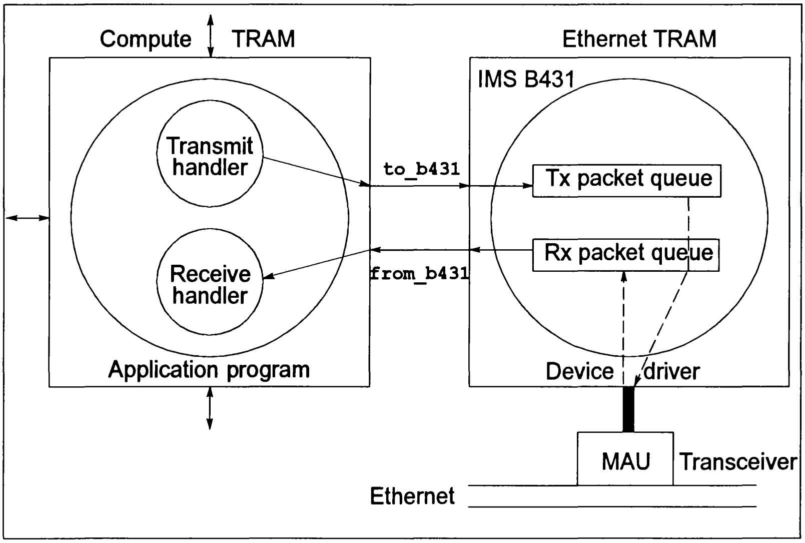

Figure 3.1 Software elements for active ethernet

Statistics concerning operation of the Ethernet interface are gathered by the device driver. They provide useful information relating to the performance and loading characteristics of the Ethernet network. See Section 5.1.5 for a description of these values. Application programs may request the current set of accumulated statistics, from the device driver, at any time.

In order to establish confidence in the correct operation of the Ethernet interface the device driver may be requested to perform diagnostic tests on the hardware it controls. Testing occurs by transmitting test packets which are looped backed either within the Ethernet interface hardware itself or via the attached network. An additional diagnostic feature, which may be enabled during Ethernet interface initialisation, tests for correct operation of the transmit and receive packet buffer memory.

3.3 IMS F006A interface libraries

The IMS F006A ANSI C and occam interface libraries contain an equivalent set of procedures. Procedures are provided to:–

- Initialise, start or stop the Ethernet interface
- Perform diagnostic loopback tests
- Transmit or receive IEEE 802.3 Ethernet packets
- Request or reset Ethernet statistics
- Download the IMS B431 device driver via an *EDGE* link

Details of the interaction between the IMS B431 device driver and client application programs are hidden by the use of these procedures. Application programs supply parameters which are packaged up and sent to the device driver for processing, data received in the opposite direction is unpacked and returned to the application program via a suitable procedure call.

The procedures for transmitting and receiving Ethernet packets are designed such that they may be called from two separate, parallel, processes. In ANSI C this is achieved by forking a new process with the `ProcAlloc()` and `ProcRun()` functions, while in occam the `PAR` construct achieves the same effect.

3.4 Example programs and source code

Example programs written in ANSI C and occam that demonstrate the use of the IMS F006A interface libraries are provided in the directories `\F006A\CLIB\EXAMPLES` and `\F006A\OCCAMLIB\EXAMPLES`. Configuration files are also included which describe a fairly typical transputer network consisting of a single IMS B431 Ethernet TRAM and a general purpose compute TRAM (which executes each example program). These can easily be modified to suite more specifc hardware if necessary.

Source code of both the ANSI C and occam interface libraries are provided in the directories `\F006A\CLIB\SOURCE` and `\F006A\OCCAMLIB\SOURCE` respectively. Most of the procedures are quite simple. The source code is supplied to demonstrate the low level interaction required to control the device driver from an adjacent transputer, this should allow similar interfaces to be produced for other, non-INMOS toolset, environments.

The files `imsb431.h` and `imsb431.inc`, also located in the source directories, contain ANSI C and occam definitions of a constant byte array. This is a bootable version of the IMS B431 device driver. Copying the contents of the array down a transputer link connected to a previously reset IMS B431 Ethernet TRAM will boot the TRAM with device driver software. This too allows alternative schemes to be developed for using the IMS B431 in other systems environments.

3.5 Environments

Transputer programs that incorporate the IMS F006A software and target a particular transputer network topology are constructed, using INMOS toolset utilities, in the conventional manner:- top level programs for each transputer in the target network are compiled, linked and configured into a single bootable file for the network. No software, other than the IMS B431 device driver, may execute on the IMS B431 TRAM. The device driver is supplied in linked unit format and should be placed on each IMS B431 TRAM in the network using the configurer tool. Because the device driver uses a single transputer link each IMS B431 TRAM will be physically connected to only one other transputer in the network. The remainder of the network is configured to run application software.

A minimal target transputer system will consist of at least an IMS B431 Ethernet TRAM and a compute TRAM. Larger systems may contain any number and combination of Ethernet and compute TRAMs, depending on the intended application. Embedded systems will probably also have a ROM TRAM to act as a bootstrap master, this will contain code to be booted into the transputer network at system startup time. IMS F006A, when used in conjunction with an INMOS toolset, permits the development of both embedded and hosted systems.

3.5.1 Development Environment

A typical hosted development environment is shown in the diagram below. It consists of an IBM PC/AT or compatible host computer installed with a TRAM motherboard, the motherboard is fitted with an IMS B431 Ethernet TRAM and a compute TRAM. The compute TRAM is used to run INMOS tools during development and the application program during test phases. The Ethernet TRAM executes the device driver. Interaction between the device driver and application program occurs over the hard link connecting both TRAMs.

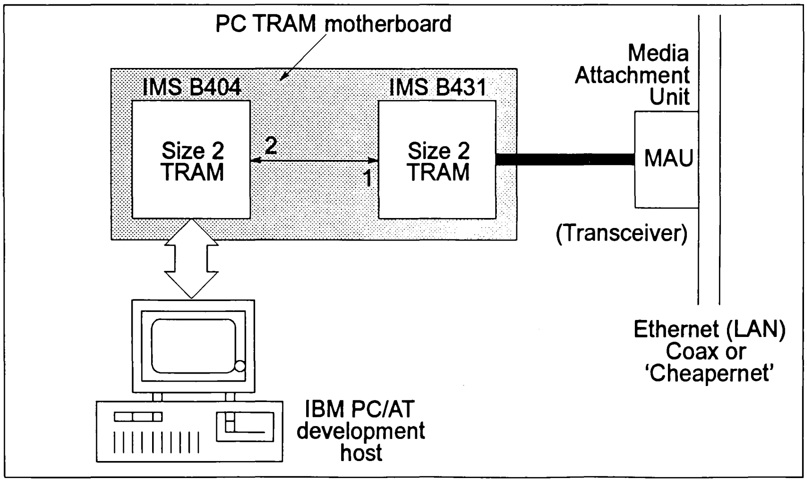

Figure 3.2 Example development environment

3.5.2 Target System Environment

The following diagram shows an example embedded system. The ROM TRAM contains configured application code including copies of the IMS B431 device driver which it boots into the transputer network. In this example the target transputer network consists of a single compute TRAM and two IMS B431 Ethernet TRAMs.

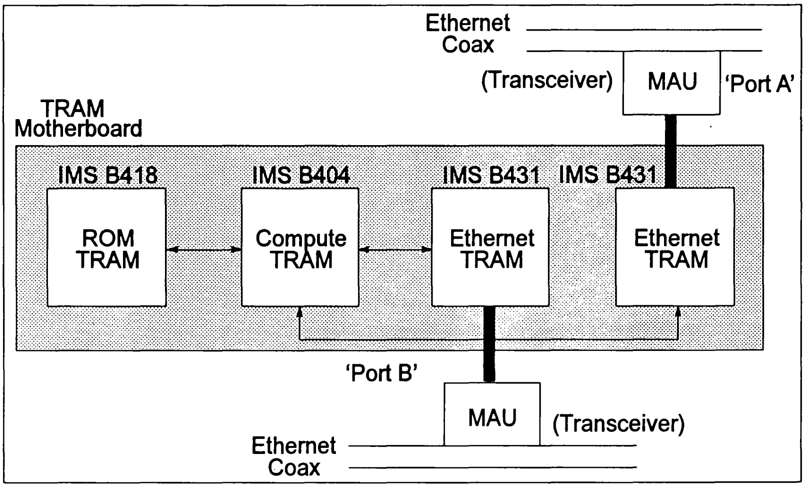

Figure 3.3 Example embedded system featuring IMS B431

4 IMS F006A libraries

To develop with the IMS F006A interface libraries:

ANSI C programmers should:–

1 Add `\F006A\;\F006A\CLIB\;` to the `ISEARCH` environment variable.

2 `#include` the header file `b431io.h`. Also include `b431test.h` if using the diagnostic loopback functions provided with the ANSI C interface library.

3 Link against the library `b431.lib`.

4 Configure the target transputer network:- place the IMS B431 device driver on the IMS B431 TRAM and application software on other TRAMs.

occam programmers should:–

1 Add `\F006A\;\F006A\OCCAMLIB\;` to the `ISEARCH` environment variable.

2 `#INCLUDE` the header file `b431io.inc`. Also include `b431test.inc` if using the diagnostic loopback procedures provided with the occam interface library.

3 `#USE` and link against the library `B431.LIB`.

4 Configure the transputer network:- place the IMS B431 device driver on the IMS B431 TRAM and application software on other TRAMs.

4.1 Interface procedures

This section describes the procedures used to interact with the IMS B431 device driver. Interaction with the device driver occurs in one of two modes, which reflect the state of the Ethernet interface it controls, as follows:–

- The Ethernet interface is inactive (stopped), it ignores all activity on the Ethernet network. Procedures are provided to initialise and then start the interface. The device driver arranges for the Ethernet interface to be in this state at startup time.
- The Ethernet interface is active (started), packet I/O proceeds concurrently. Procedures are provided to transmit and receive packets, request Ethernet statistics or revert the interface to the inactive state.

The Ethernet interface must be initialised before it can be started. Once started, the application may fork into the separate parallel processes, described earlier, to handle packet I/O. The device driver will attempt to deliver packets it receives from the Ethernet interface on the `from_b431` channel, `B431_Waitfor_Event()` should be called to wait for and receive them. `B431_Tx_Packet1()` or `B431_Tx_Packet2()` are used to transmit packets by sending them to the device driver along the `to_b431` channel. `B431_Waitfor_Event()` is also used to accept other events from the device driver such as notification of error conditions or Ethernet statistics. The Ethernet interface may be subsequently stopped, in order to alter initialisation parameters, or during software termination.

4.1.1 B431_Init_Normal()

Description:

Initialise the Ethernet interface for normal operation (as opposed to diagnostic loopback operation).

This sets the physical address and logical address filter (multicast address group) for the Ethernet interface. Packets transmitted by the Ethernet interface will carry a source address equal to the physical address assigned by calling this procedure. Packets will be received if they carry a destination address equal to either the physical address or a multicast destination address matched by the logical address filter (or a broadcast address). Section 5.1.3 describes Ethernet addressing in detail.

Normal operation can be modified by enabling a number of optional mode flags.

Once initialised, the Ethernet interface can be started, see `B431_Start_Ether()`.

C:

```
int B431_Init_Normal(
Channel               *from_b431,
Channel               *to_b431,
const unsigned char   physical_address[PHYSICAL_ADDRESS_SIZE],
const unsigned char   logical_address_filter[LOGICAL_ADDRESS_FILTER_SIZE],
const long int        mode_flags )
```

occam:

```
PROC B431.Init.Normal(
CHAN OF ANY                                from.b431,
                                           to.b431,
VAL [PHYSICAL.ADDRESS.SIZE]BYTE            physical.address,
VAL [LOGICAL.ADDRESS.FILTER.SIZE]BYTE logical.address.filter,
VAL INT                                    mode.flags,
BYTE                                       result )
```

Parameter	Comments
`from_b431`	Channel from the B431 device driver
`to_b431`	Channel to the B431 device driver
`physical_address`	Physical address of the Ethernet interface
`logical_address_filter`	Logical address filter for the Ethernet interface
`mode_flags`	Mode flags bit mask

Mode flags:

`mode_flags` is a bit mask. Bits are set to enable optional Ethernet interface functions, or to modify the normal operating mode. The following bit masks are defined:–

`MEMORY_CHECK`	Enable execution of the packet buffer memory check
`PROMISCUOUS_RX`	Enable promiscuous mode packet reception. All packets will be received, regardless of their destination address, see Section 5.1.3.
`MONITOR_HEARTBEAT`	Enable heartbeat monitoring, see Section 5.1.7.
`DISABLE_TX_CRC`	Disable automatic transmit packet CRC field generation, see Section 5.1.2.
`DISABLE_TX_RETRY`	Disable transmitter retries, see Section 5.1.4.

Return codes:

`result` contains the completion code for `B431.Init.Normal()`. It is returned by `B431_Init_Normal()`.

A value of `INIT_SUCCESS` indicates normal successful completion. Other result codes indicate a failure to initialise the Ethernet interface as follows:–

`INIT_NOT_STOPPED`	The Ethernet interface is active. It should be in the stopped state, see `B431_Stop_Ether()`
`INIT_HARDWARE_FAILED`	The Ethernet interface hardware failed to initialise correctly
`INIT_MEMORY_FAULT`	A memory fault was discovered while executing the packet buffer memory check.

Notes:

If automatic CRC field generation is disabled then packets supplied to the device driver for transmission should contain a user supplied CRC field. Section 5.1.2 describes the algorithm specified in the IEEE 802.3 CSMA/CD Ethernet standard for calculating the CRC value.

The heartbeat monitor function should only be enabled if the Media Attachment Unit (MAU) generates the heartbeat signal. This is usually a jumper configured option within the MAU (transceiver) box.

4.1.2 B431_Init_Loopback()

Description:

Initialise the Ethernet interface for loopback operation. When operating in loopback mode the Ethernet interface will return any self addressed transmit packets as received packets. This can occur within the Ethernet interface itself:- internal loopback, or via the network:- external loopback.

Once initialised, the Ethernet interface can be started, see `B431_Start_Ether()`.

Loopback operation is intended for diagnostic purposes only. Application programs will normally use the supplied diagnostic procedures, described in Section 4.2, to perform diagnostic tests and will not call this procedure. It is provided in the interface library for users that wish to write their own diagnostic software.

C:

```
int B431_Init_Loopback(
Channel              *from_b431,
Channel              *to_b431,
const unsigned char  physical_address[PHYSICAL_ADDRESS_SIZE],
const long int       mode_flags )
```

occam:

```
PROC B431.Init.Loopback(
CHAN OF ANY                        from.b431,
                                   to.b431,
VAL [PHYSICAL.ADDRESS.SIZE]BYTE physical.address,
VAL INT                            mode.flags,
BYTE                               result )
```

Parameter	Comments
`from_b431`	Channel from the B431 device driver
`to_b431`	Channel to the B431 device driver
`physical_address`	Physical address of the Ethernet interface
`mode_flags`	Mode flags bit mask

Mode flags:

`mode_flags` is a bit mask. Bits are set to enable optional Ethernet interface functions, or to modify the loopback operating mode. The following bit masks are defined:–

`MEMORY_CHECK`	Enable execution of the packet buffer memory check
`INTERNAL_LOOPBACK`	Enable internal loopback mode, default is external mode
`FORCE_COLLISION`	Enable transmit collisions, packets will be forced to collide. This tests the collision detection logic. Only valid in internal loopback mode
`MONITOR_HEARTBEAT`	Enable heartbeat monitoring, see Section 5.1.7.
`DISABLE_TX_CRC`	Disable automatic transmit packet CRC field generation, see Section 5.1.2.
`DISABLE_TX_RETRY`	Disable transmitter retries, see Section 5.1.4.

Return codes:

`result` contains the completion code for `B431.Init.loopback()`. It is returned by `B431_Init_loopback()`.

A value of `INIT_SUCCESS` indicates normal successful completion. Other result codes indicate a failure to initialise the Ethernet interface as follows:–

`INIT_NOT_STOPPED`	The Ethernet interface is active. It should be in the stopped state, see `B431_Stop_Ether()`
`INIT_HARDWARE_FAILED`	The Ethernet interface hardware failed to initialise correctly
`INIT_MEMORY_FAULT`	A memory fault was discovered while executing the packet buffer memory check

Notes

If automatic CRC field generation is disabled then packets supplied to the device driver for transmission should contain a user supplied CRC field. Section 5.1.2 describes the algorithm specified in the IEEE 802.3 CSMA/CD Ethernet standard for calculating the CRC value.

The heartbeat monitor function should only be enabled if the Media Attachment Unit (MAU) generates the heartbeat signal. This is usually a jumper configured option within the MAU (transceiver) box.

4.1.3 B431_Start_Ether()

Description:

Start the Ethernet interface. Packet I/O will be enabled.

The Ethernet interface must be initialised before it is started, see `B431_Init_Normal()`.

C:

```
int B431_Start_Ether(
Channel *from_b431
Channel *to_b431 )
```

occam:

```
PROC B431.Start.Ether(
CHAN OF ANY from.b431,
            to.b431,
BYTE        result )
```

Parameter	Description
`from_b431`	Channel from the B431 device driver
`to_b431`	Channel to the B431 device driver

Return codes:

`result` contains the completion code for `B431.Start.Ether()`. It is returned by `B431_Start_Ether()`.

A value of `START_SUCCESS` indicates normal successful completion. Other result codes indicate a failure to start the Ethernet interface as follows:–

`NO_INIT_DONE`	The Ethernet interface has not been initialised

Notes:

Once the Ethernet interface has started, the channels connecting the application program to the B431 device driver (`to_b431` and `from_b431`) become asynchronous. Commands to the device driver are issued by calling procedures with the `to_b431` channel only. Eg:– `B431_Tx_Packet1()` or `B431_Ether_Stats()`. The device driver will respond (if necessary) by sending results on `from_b431`. `B431_Waitfor_Event()` is used to wait for and decode events generated by the device driver on this channel, three types of event can be expected:–

1 Received packets.

2 Report of an error condition.

3 A response to `B431_Stop_Ether()`, `B431_Ether_Stats()` or `B431_Terminate()`.

Application software should fork into two separate parallel processes after calling `B431_Start_Ether()`. One process will be responsible for transmitting packets and sending commands to the device driver, the other will continuously loop calling `B431_Waitfor_Event()` to receive packets or command responses. Source code examples of this are described in section B.1.1 for ANSI C, and section B.1.2 for occam.

4.1.4 B431_Tx_Packet1()

Description:

Transmit an Ethernet packet. The packet is sent to the device driver where it is buffered and then transmitted on the Ethernet. Packets are buffered in a first in first out queue (fifo) where they wait to be transmitted. If the queue fills up then this procedure will block until there is sufficient space for the new packet. This behaviour does not affect the receive packet queue, which will continue to supply packets, if any are received. See `B431_Waitfor_Event()`.

C:

```
void B431_Tx_Packet1(
Channel                 *to_b431,
const unsigned char     *ethernet_packet,
const int               ethernet_packet_length )
```

occam:

```
PROC B431.Tx.Packet1(
CHAN OF ANY to.b431,
VAL []BYTE  ethernet.packet )
```

Parameter	Description
`to_b431`	Channel to the B431 device driver
`ethernet_packet`	Ethernet packet
`ethernt_packet_length`	Packet length

Notes:

`ethernet_packet` should contain a header (see Section 5.1.1). This will contain a destination address, source address and type length field. The source address should correspond to the address assigned to the Ethernet interface when initialised. If the Ethernet interface was initialised with transmit CRC field generation disabled then `ethernet_packet` should also contain a user supplied CRC field.

In normal operation the minimum packet length is `MIN_PACKET_LENGTH` (64) bytes, and the maximum length is `MAX_PACKET_LENGTH` (1518) bytes. These values include the leading 14 byte packet header and trailing 4 byte CRC field. If automatic CRC field generation is disabled then the minimum and maximum packet lengths will be 4 bytes less. Packets containing less than the minimum number of bytes will be padded automatically to the minimum length.

4.1.5 B431_Tx_Packet2()

Description:

Transmit an Ethernet packet. The packet header and data segments are sent to the device driver where they are assembled into a complete packet, buffered, and then transmitted on the Ethernet. The device driver will insert the source Ethernet address into the packet header automatically, this is the address assigned to the Ethernet interface during initialisation. Packets are buffered in a first in first out queue (fifo) where they wait to be transmitted. If the queue fills up then this procedure will block until there is sufficient space for the new packet. This behaviour does not affect the receive packet queue, which will continue to supply packets, if any are received. See `B431_Waitfor_Event()`.

C:

```
void B431_Tx_Packet2(
Channel              *to_b431,
const unsigned char  destination_address[PHYSICAL_ADDRESS_SIZE],
const short int      type_length_field,
const unsigned char  *packet_data[],
const int            packet_data_length )
```

occam:

```
PROC B431.Tx.Packet2(
CHAN OF ANY                           to.b431,
VAL [PHYSICAL.ADDRESS.SIZE]BYTE destination.address,
VAL INT16                             type.length.field,
VAL []BYTE                            packet.data )
```

Parameter	Description
`to_b431`	Channel to the B431 device driver
`destination_address`	Destination Ethernet address
`type_length_field`	Type or length value, the least significant byte corresponds to the most significant byte of the type length field
`packet_data`	Packet data
`packet_data_length`	Data length

Notes:

If the Ethernet interface was initialised with transmit CRC field generation disabled then `packet_data` should also contain a user supplied CRC field.

In normal operation the minimum packet length is `MIN_PACKET_LENGTH` (64) bytes, and the maximum length is `MAX_PACKET_LENGTH` (1518) bytes. These values include the leading 14 byte packet header and trailing 4 byte CRC field. If automatic CRC field generation is disabled then the minimum and maximum packet lengths will be 4 bytes less. Packets containing less than the minimum number of bytes will be padded automatically to the minimum length.

4.1.6 B431_Reset_Stats()

Description:

Reset the Ethernet statistics. The device driver will clear (reset to 0) it's accumulated Ethernet statistics.

C:

```
void B431_Reset_Stats( Channel *to_b431 )
```

occam:

```
PROC B431.Reset.Stats( CHAN OF ANY to.b431 )
```

Parameter	Description
`to_b431`	Channel to the B431 device driver

4.1.7 B431_Stop_Ether()

Description:

Stop the Ethernet interface. Packet transmission and reception will cease and the Ethernet interface will be disabled. Any queued transmit or receive packets will be discarded. When the Ethernet interface has stopped the device driver will send an acknowledge, `B431_Waitfor_Event()` should be called to receive the acknowledgement. Once stopped the Ethernet interface can be re-initialised or re-started.

C:

```
void B431_Stop_Ether( Channel *to_b431 )
```

occam:

```
PROC B431.Stop.Ether( CHAN OF ANY to.b431 )
```

Parameter	Description
`to_b431`	Channel to the B431 device driver

4.1.8 B431_Terminate()

Description:

Terminate the B431 device driver. The device driver will acknowledge and then terminate. `B431_Waitfor_Event()` should be called to receive the acknowledgement. Once terminated no further interaction with the device driver is possible.

C:

```
void B431_Terminate( Channel *to_b431 )
```

occam:

```
PROC B431.Terminate( CHAN OF ANY to.b431 )
```

Parameter	Description
`to_b431`	Channel to the B431 device driver

4.1.9 B431_Ether_Stats()

Description:

Request the Ethernet statistics. The device driver will return it's current set of accumulated Ethernet statistics, `B431_Waitfor_Event()` should be called to receive them. Section 5.1.5 describes the statistics returned.

C:

```
void B431_Ether_Stats( Channel *to_b431 )
```

occam:

```
PROC B431.Ether.Stats( CHAN OF ANY to.b431 )
```

Parameter	Description
`to_b431`	Channel to the B431 device driver

4.1.10 B431_Waitfor_Event()

Description:

Wait for an Ethernet interface event. This procedure is called to wait for and decode events from the device driver when the Ethernet interface is active (see `B431_Start_Ether()`). The device driver will generate three types of event:–

1 Received packets

2 Report of an error condition

3 A response to `B431_Stop_Ether()`, `B431_Ether_Stats()` or `B431_Terminate()`.

`B431_Waitfor_Event()` is designed to be called from a process primarily responsible for handling the reception of Ethernet packets. Another, parallel process, will handle the transmission of packets independently of this.

C:

```
int B431_Waitfor_Event(
Channel          *from_b431,
Channel          *cancel,
ETHER_STATS      *ethernet_stats,
unsigned char    *ethernet_packet,
int              *ethernet_packet_length,
int              *error_code,
unsigned char    *failed_packet_data )
```

occam:

```
PROC B431.Waitfor.Event(
CHAN OF ANY                  from.b431,
CHAN OF BYTE                 cancel,
[ETHER.STATS.SIZE]INT32      ethernet.stats,
[MAX.PACKET.LENGTH]BYTE      ethernet.packet,
INT                          ethernet.packet.length,
                             error.code,
[FAILED.PACKET.LENGTH]BYTE failed.packet.data,
BYTE                         result )
```

Parameter	Description
`from_b431`	Channel from the B431 device driver
`cancel`	Cancel channel, used to force completion of the procedure call. Sending any byte value on this channel will cause the procedure to return immediately
`ethernet_stats`	Ethernet statistics, an unsigned long integer structure (INT32 array in occam) that contains accumulated Ethernet statistics. Only valid if `result` is `B431_ETHER_STATS`
`ethernet_packet`	Received Ethernet packet, includes trailing CRC field. Only valid if `result` is `B431_RX_PACKET`
`ethernet_packet_length`	Packet length
`error_code`	Error code, describes an error event. Only valid if `result` is `B431_ERROR_REPORT`
`failed_packet_data`	Failed packet data, contains the first `FAILED_PACKET_LENGTH` bytes of a failed transmit packet. Only valid if `error_code` is `ERROR_TX_PACKET_FAILED`

Return codes:

`result` contains the completion code for `B431.Waitfor.Event()`. It is returned by `B431_Waitfor_Event()`.

The `result` value indicates the reason for completion, as follows:–

Parameter	Description
`B431_WAIT_CANCELLED`	Indicates forced completion with the cancel channel
`B431_TERMINATE`	Acknowledgement of device driver termination. See `B431_Terminate()`, `B431_Waitfor_Event()` should not be called again
`B431_STOP_ETHER`	Acknowledgement of an Ethernet interface stop request. See `B431_Stop_Ether()`
`B431_ETHER_STATS`	Acknowledgement of a request for Ethernet statistics. See `B431_Ether_Stats()`, `ethernet_stats` will contain the current, accumulated, Ethernet statistics
`B431_RX_PACKET`	Indicates reception of an Ethernet packet. `ethernet_packet` and `ethernet_packet_length` will hold the packet and length of packet respectively
`B431_ERROR_REPORT`	Indicates an occurence of an internal or Ethernet related error condition. `error_code` will contain a reason code, see below for these

`error_code` will contain an error reason code (if `result` is set to `B431_ERROR_REPORT`), as follows:–

Parameter	Description
`ERROR_NO_ERROR`	No error
`ERROR_TX_PACKET_FAILED`	Device driver failed to transmit a packet after the maximum number of transmit retries was attempted. The first `FAILED_PACKET_LENGTH` bytes of the failed packet is returned
`ERROR_HEARTBEAT_STOPPED`	Indicates a failure to detect the heartbeat signal from the Media Attachment Unit following packet transmission. See Section 5.1.7.
`ERROR_TX_BABBLE_FAULT`	Indicates an attempt to transmit a packet longer than the maximum allowed, `MAX_PACKET_LENGTH` is 1518 bytes
`ERROR_DMA_REQUEST_LATE`	Fatal hardware fault, this should never occur
`ERROR_TX_BUFFER_INVALID`	Fatal internal condition, this should never occur
`ERROR_TX_BUFFER_UNDERFLOW`	Fatal hardware fault, this should never occur

4.2 Diagnostic procedures

Two diagnostic procedures are provided for verifying correct operation of the Ethernet interface hardware. The tests performed by both procedures involve transmitting self addressed packets, waiting for them to return and then comparing the returned packet data with the original test data. This loopback is either confined to the bounds of the Ethernet interface hardware itself (internal) or may occur externally, via an attached network.

Internal loopback checks for correct operation of the Ethernet interface hardware and associated packet buffer memory while transmitting and receiving test packets. There is no interaction with the attached Ethernet network during the test. A failure to successfully perform internal loopback indicates a fatal fault within the Ethernet interface hardware. External loopback extends the scope of the tests performed by transmitting and receiving test packets via the live Ethernet. Failures during this test can be caused by problems with the Media Attachment Unit, associated cabling or a network fault.

Since both procedures initialise the Ethernet interface in diagnostic loopback mode, application software should re-initialise the interface after their use. The Ethernet interface will be left in the inactive (stopped) state.

4.2.1 B431_Internal_Loopback()

Description:

Perform an internal loopback test of the Ethernet interface hardware.

C:

```
int B431_Internal_Loopback(
Channel                 *from_b431,
Channel                 *to_b431,
const unsigned char     physical_address[PHYSICAL_ADDRESS_SIZE],
int                     *error_code )
```

occam:

```
PROC B431.Internal.Loopback(
CHAN OF ANY                        from.b431,
                                   to.b431,
VAL [PHYSICAL.ADDRESS.SIZE]BYTE physical.address,
INT                                result,
                                   error.code )
```

Parameter	Description
`from_b431`	Channel from the B431 device driver
`to_b431`	Channel to the B431 device driver
`physical_address`	Physical address of the Ethernet interface
`error_code`	An Ethernet interface error code if `result` is `TEST_HARDWARE_ERROR`

Return codes:

`result` contains the completion code for `B431.Internal.Loopback()`. It is returned by `B431_Internal_Loopback()`.

A value of `TEST_PASSED` indicates normal successful completion:– The Ethernet interface successfully passed the packet loopback test. Other result codes indicate a failure to perform the loopack test as follows:–

`TEST_INIT_FAILED`	The Ethernet interface hardware failed to initialise correctly
`TEST_MEMORY_FAULT`	A memory fault was discovered while executing the packet buffer memory check
`TEST_LOOPBACK_FAILED`	Test packets failed to return or were corrupt on reception
`TEST_HARDWARE_ERROR`	The device driver indicated a hardware fault, `error_code` will be as described for `B431_Waitfor_Event()`, see section 4.1.10.

Notes:

Since no interaction with the external Ethernet network occurs during internal loopback the physical address assigned to the Ethernet interface may or may not be a valid address.

4.2.2 B431_External_Loopback()

Description:

Perform an external loopback test of the Ethernet interface hardware.

C:

```
int B431_External_Loopback(
Channel              *from_b431,
Channel              *to_b431,
const unsigned char  physical_address[PHYSICAL_ADDRESS_SIZE],
int                  *error_code )
```

occam:

```
PROC B431.External.Loopback(
CHAN OF ANY                          from.b431,
                                     to.b431,
VAL [PHYSICAL.ADDRESS.SIZE]BYTE physical.address,
INT                                  result,
                                     error.code )
```

Parameter	Description
`from_b431`	Channel from the B431 device driver
`to_b431`	Channel to the B431 device driver
`physical_address`	Physical address of the Ethernet interface
`error_code`	An Ethernet interface error code if `result` is `TEST_HARDWARE_ERROR`

Return codes:

`result` contains the completion code for `B431.External.Loopback()`. It is returned by `B431_External_Loopback()`.

A value of `TEST_PASSED` indicates normal successful completion:– The Ethernet interface successfully passed the packet loopback test. Other result codes indicate a failure to perform the loopack test as follows:–

`TEST_INIT_FAILED`	The Ethernet interface hardware failed to initialise correctly
`TEST_MEMORY_FAULT`	A memory fault was discovered while executing the packet buffer memory check
`TEST_LOOPBACK_FAILED`	Test packets failed to return or were corrupt on reception
`TEST_HARDWARE_ERROR`	The device driver indicated a hardware fault, `error_code` will be as described for `B431_Waitfor_Event()`, see section 4.1.10.

4.3 IMS B431 Device Driver

The IMS B431 device driver is supplied as a linked unit in the file:– `\F006A\B431DRVR.LKU` (which should be on the `ISEARCH` path). It is referenced by either a C (`.cfs`) or an occam (`.pgm`) configuration level program and should be placed on every IMS B431 Ethernet TRAM in the target transputer network. The configuration language syntax for achieving this is different for ANSI C and occam, the following configuration program source code fragments summarise this:–

occam `.pgm` file:

```
-- Hardware description

VAL K IS 1024 :

NODE b431 :
NETWORK simple.network
  DO
    -- IMS B431 has a T222 and 64 Kbytes memory
    SET b431 (type, memsize := "T212", 64 * K)
    CONNECT SomeOtherTram[link][2] TO b431[link][1]
    -- Can use an arbitrary link on IMS B431
:

-- Software description

#USE "b431drvr.lku"  -- IMS B431 device driver

CONFIG
  CHAN OF ANY SomeOtherTram.to.b431, b431.to.SomeOtherTram :
  PLACED PAR
    PROCESSOR b431
      INMOS.B431.Driver( SomeOtherTram.to.b431, b431.to.SomeOtherTram )
 :
```

ANSI C `.cfs` file:

```
/* Hardware description */

/* IMS B431 has a T222 and 64 Kbytes memory */
T212 (memory = 64K) b431;
connect SomeOtherTram.link[2] to b431.link[1];

/* Software description */

process ( interface ( input to_b431, output from_b431 ) ) b431_driver;

connect SomeOtherTram.to_b431 to b431_driver.to_b431;
connect SomeOtherTram.from_b431 to b431_driver.from_b431;

/* Select linked units and place processes */

use "b431drvr.lku" for b431_driver;
/* IMS B431 device driver */

place b431_driver on b431;
```

4.3.1 Debugging support

To achieve maximum performance from the Ethernet interface the IMS B431 device driver linked unit `b431drvr.lku` is supplied with interactive debugging disabled. Note that it is **not** supplied with interactive debugging enabled, for the following reasons:–

1 The IMS B431 device driver will be used in real time communication system environments. Typically, it is not possible to interactively debug real time communication protocol software.

2 Because it would be slower, using a different and slower version of the device driver when debugging has the potential to alter the conditions under which bugs arise.

The consequence of this is that it is **not** possible to interactively debug transputer networks whose configuration descriptions include placement statements for the IMS B431 device driver. In particular, for ANSI C programmers, this means:–

1 When booted with `ISERVER`, configured programs will run normally.

2 Attempting to boot a configured program with the interactive debugger `IDEBUG` will fail, the debugger will generate a warning message.

3 Application debugging **is** possible with the post-mortem debugger.

and for occam programmers:–

1 Application software should be compiled and linked with interactive debugging disabled. Use the complier and linker `y` option.

2 The `y` option should also be used when configuring with `occonf`, it will fail to generate a bootable otherwise.

3 When booted with `ISERVER`, configured programs will run normally.

4 Application debugging **is** possible with the post-mortem debugger.

Interactive debugging **is** possible with the IMS F006A software, but this requires alteration to source code and network configuration descriptions. There will be occasions when interactive debug capability can be usefully applied to some parts of a target transputer network, while other parts continue to execute real time software. With care, it may even be possible to achieve this without affecting the real time nature of other transputers in the system. For this reason an alternative scheme for booting a target transputer network that allows both the interactive debugging of application software and the IMS B431 device driver to execute is supported by the IMS F006A software.

Every IMS B431 Ethernet TRAM, in a target network, is treated as a peripheral device and is not described in the corresponding network configuration file. The placed hard links and channels that previously connected application software to the device driver are instead placed onto an *EDGE* link. This separates application transputers from their attached IMS B431 Ethernet TRAMs. No bootable code will be generated for the IMS B431 TRAMs because they will not appear in the network description:– the network bootable file generated by configuring with the alternative description will not contain any code for any IMS B431 TRAMs. When the network is booted with this file it will run only application software, on application transputers. The IMS B431 TRAMs will be reset and not running any code.

The responsibility to boot each IMS B431 TRAM in this scheme is passed on to application software. The procedure `B431_Load_Driver()` does this, when passed a pair of transputer channels which are connected to an adjacent IMS B431 TRAM it will download the device driver into the TRAM and verify that the boot succeeded. The channels will be the same ones subsequently used to interact with the device driver, as previously described, and should be mapped onto an EDGE link known to be physically connected to the IMS B431 TRAM.

Booting the IMS B431 TRAM separately in this manner allows an application transputer network to be run with or without the interactive debugger present whilst maintaining constant behaviour on the Ethernet TRAM. It should be stressed, however, that the use of the interactive debugger **does** change the performance and therefore real time characteristics of the application being debugged. It is recommended that the post mortem debugger be used unless special circumstances require otherwise.

The following occam configuration source code fragment shows how to declare an EDGE link and how they are then attached to an application program:–

```
-- Hardware description.

VAL K IS 1024 :
VAL M IS K * K :

EDGE b431 :
ARC HostLink, B431Link :
NODE app :
NETWORK application
  DO
    -- Application code runs on an IMS B404 2Mbye TRAM
    SET app (type, memsize := "T800", 2 * M)
    CONNECT app[link][1] TO HOST WITH HostLink
    -- IMS B404 TRAM is known to be connected to the IMS B431
    -- via it's link 2. It doesn't matter to which link on the
    -- IMS B431 TRAM it is connected.
    CONNECT app[link][2] TO b431 WITH B431Link
:
```

```
-- Software description.

#INCLUDE "hostio.inc"
#USE "application.c8h"  -- Application program

CONFIG
  CHAN OF SP app.to.host, host.to.app :
  CHAN OF ANY app.to.b431, b431.to.app :
  PLACE app.to.host, host.to.app ON HostLink :
  PLACE app.to.b431, b431.to.app ON B431Link :
  PLACED PAR
    PROCESSOR root
      application( host.to.app, app.to.host, b431.to.app, app.to.b431)
:
```

4.4 Using B431_Load_Driver()

Description:

Load the IMS B431 Ethernet TRAM with its device driver.

Applications, which do not include the IMS B431 Ethernet TRAM in their transputer network configuration, will use this procedure to download the device driver. The entire transputer network, with the exception of the Ethernet TRAM, will be loaded with application software at bootstrap time. This procedure allows the application to subsequently download the device driver into the Ethernet TRAM. Once loaded, the Ethernet TRAM will function exactly as it would as if booted with the device driver from the same configured bootable file as the application. This alternative (two stage) loading scheme is required when one part of the transputer network needs to be configured differently from another part. Section 4.3.1 describes this in more detail.

C:

```
#include <b431load.h>

int B431_Load_Driver(
        Channel *from_b431
        Channel *to_b431 )
```

occam:

```
#INCLUDE "b431load.inc"

PROC B431.Load.Driver(
  CHAN OF ANY from.b431,
              to.b431,
  BYTE        result )
```

Parameter	Description
`from_b431`	Channel from the B431 device driver
`to_b431`	Channel to the B431 device driver

Return codes:

`result` contains the completion code for `B431.Load.Driver()`. It is returned by `B431_Load_Driver()`.

A value of `LOAD_SUCCESS` indicates normal successful completion:– the IMS B431 Ethernet TRAM will be running it's device driver. Other result codes indicate a failure to load and start the device driver as follows:–

`LOAD_BOOT_FAILED`	Could not download the device driver
`LOAD_CHECK_FAILED`	Loaded device driver but it failed to respond when checked

Notes:

`from_b431` and `to_b431` must be configured as `EDGE` channels, ie: they are placed onto an `EDGE` link by the configuration description for the application transputer network. The corresponding hard link should connect the EDGE to an IMS B431 Ethernet TRAM. This is discussed in the relevant toolset documentation for the configurer tools:- `icconf` (ANSI C) or `occonf` (occam).

If the EDGE link is physically unconnected, or connected to something other than an IMS B431 Ethernet TRAM, then this procedure will fail and interaction with the device driver will be impossible.

5 IEEE 802.3 CSMA/CD Ethernets

5.1 IEEE 802.3 CSMA/CD Ethernets

Ethernet is a Local Area Network (LAN) standard originally proposed by Xerox Corp, Digitial Equipment Corp and Intel Corp. It is now established as ANSI/IEEE Standard IEEE 802.3. Cheapernet is a variant of Ethernet using the same signalling conventions but a cheaper network medium.

Ethernet is a Carrier Sense, Multiple Access, with Collision Detection local area network (abbreviated to CSMA/CD). This term defines the way in which two or more nodes gain and share access to a common bus transmission medium:- the Ethernet coaxial cable.

Communication is by means of packets. A packet is a collection of octets (bytes). Packets are transmitted in byte serial order, and within each byte bit serial order, on the coaxial cable.

When a node wishes to send a packet to another node, it observes the network to see whether any other node is transmitting a packet (Carrier Sense). If it detects another transmission, it will not attempt to transmit and instead defers until the network is quite. If it does not detect another transmission it begins to transmit the packet.

Because it takes a finite time for a signal to travel across the network, two (or more) nodes may begin transmitting simultaneously because they all believe the network is free (Multiple Access). This results in packets colliding on the network.

All of the nodes detect the collision (Collision Detection). The nodes which were not attempting to transmit will then not attempt to transmit until the collision has been resolved and a packet transmitted. The nodes which were attempting to transmit wait for a random time then attempt to transmit again. Thus, one node should always gain possession of the network. If another collision occurs, the nodes wait again for a random time and try again. Up to 16 transmission attempts are allowed, if the packet has not been transmitted after this, the node does not try to transmit it again.

Because of the carrier sense function, there is only a short time at the beginning of transmission during which collisions can occur. This time is known as the slot time and is the time taken for a transmitted signal to travel to the farthest point on the network, collide, and return as a collision. The slot time is defined to be 64 byte times which at 10 Mbits/s is 51.2μs. This definition of slot time limits the maximum path length on the network to 500 meters.

Once the slot time has elapsed without any collisions being detected, a node may assume that it has possession of the network. All other nodes will have detected it's transmission and will not attempt to transmit.

5.1.1 Packet structure

Ethernet packets have the following structure:–

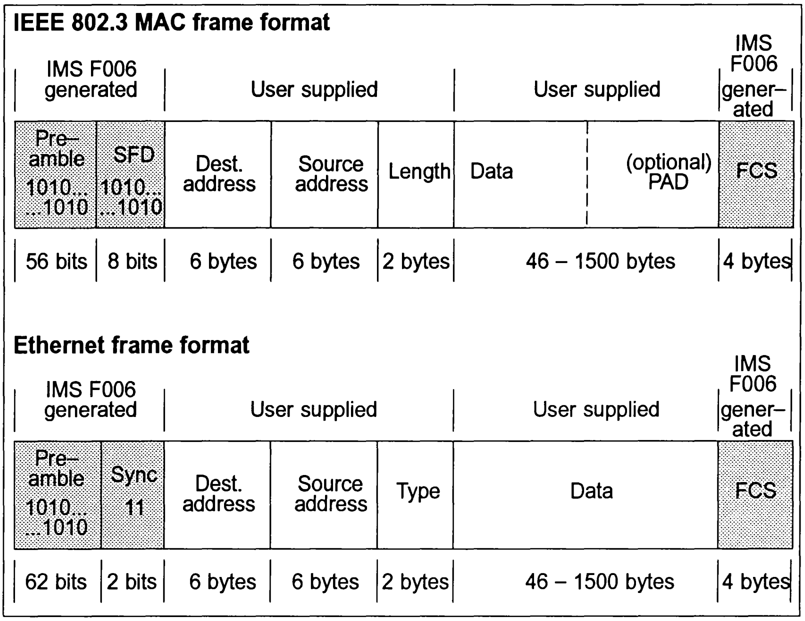

Figure 5.1 Ethernet packet structure

The IEEE 802.3 preamble and start frame delimiter (SFD) bits are equivalent to the Ethernet preamble and sync bits. Thus the packet formats are essentially identical.

Bytes are transmitted from left to right, starting with the preamble bits and finishing with the frame check sequence (FCS) bytes. The IMS B431 hardware generates the preamble and SFD / sync bits automatically, it also computes the frame check sequence bytes unless explicitly disabled from doing so.

The packet is logically split into two parts:- the packet header and data segments. The header contains addressing and control information and the packet data segment contains data.

The destination and source address fields are both 6 bytes long. The source address will correspond to the address assigned to the Ethernet interface during initialisation, it identifies the packet's origin. The destination address identifies the intended recipient of the packet. Section 5.1.3 describes Ethernet addressing conventions.

The type or length field is two bytes long, it carries either a packet type code or a length value. Type codes are used to identify packets as members of a common higher layer protocol family. When used as a length field the value corresponds to the number of bytes carried in the data part of the packet which may then be different from the actual number of bytes in the packet. This is used to distinguish data bytes from optional padding bytes. The length value should be expressed in big endian order, the most significant byte is transmitted before the least significant.

Packet lengths

The minimum Ethernet packet length is 64 (`MIN_PACKET_LENGTH`) bytes. The maximum is 1518 (`MAX_PACKET_LENGTH`) bytes. These values include the 14 (`PACKET_HEADER_SIZE`) byte packet header and trailing frame check sequence bytes. Attempting to transmit a packet longer than the maximum allowed will result in a transmit babble fault. Packets smaller than the minimum value will padded to the minimum value with random data.

5.1.2 CRC algorithm

The CRC algorithm is used to check the data integrity of packets and also as part of the logical address filtering mechanism (multicast addressing). The Ethernet interface automatically computes a 32 bit CRC value on the source address, destination address, type / length field and data segments of a packet to be transmitted. This value is appended to the packet when it is transmitted. The Ethernet interface also calculates the CRC value for received packets and compares this to the received CRC value. If they are different, the packet has been corrupted. A count of the number of packets received with CRC errors is maintained by the device driver

If the Ethernet interface is initialised with CRC field generation disabled then packets queued for transmission should have a user calculated CRC field appended to the packet data segment. See `B431_Init_Normal()`.

The CRC algorithm is specified in ANSI/IEEE 802.3 CSMA/CD, it is computed as follows:–

1 The first 32 bits of the packet are complemented.

2 The n bits of the packet are considered to the coefficients of a polynomial $M(x)$ of degree $n-1$:- the first bit of the destination address is the coefficient of the x^{n-1} term and the last bit of the data field is the coefficient of the x^0 term.

3 $M(x)$ is multiplied by x^{32} and divided by the generator polynomial:–

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

4 The remainder $R(x)$ is of degree ≤ 31. The coefficients of $R(x)$ are considered to be a 32 bit sequence.

5 The bits are complemented:- the result is the CRC.

The Ethernet interface contains hardware to compute the CRC value.

5.1.3 Addressing

Every node on an Ethernet network is assigned a unique physical address (see `B431_Init_Normal()`). Packets travelling on the Ethernet contain two physical address fields, the destination address and the source address. The source address will identify exactly one node:- the node originating the packet. The destination address may identify a single destination node, a group of nodes (multicast address) or all nodes on the Ethernet (broadcast address).

Physical addresses are 6 (`PHYSICAL_ADDRESS_SIZE`) bytes long.

Multicast Addresses

The CRC algorithm is used to implement the multicast address function. A packet carrying a destination address with a least significant bit set to 1 is a multicast packet. It may be received by a group of nodes depending on their logical address filters.

Multicast packets are filtered in the following way:–

1 The CRC algorithm (described in section 5.1.2) is applied to the multicast destination address

2 The 6 high order bits of the resultant 32 bit CRC value are used to select one bit from the logical address filter.

3 If the bit is 1 the packet is received, otherwise it is discarded.

The logical address filter is therefore 8 (`LOGICAL_ADDRESS_FILTER_SIZE`) bytes long (2^6 = 64 bits = 8 bytes).

To disable the reception of multicast packets set all 8 bytes to zero.

Broadcast Address

A packet carrying a destination address of `#FFFFFFFFFFFF`, the broadcast address, will be received by all nodes regardless of their physical or multicast address values.

5.1.4 Retry algorithm

The Ethernet interface will make up to 16 attempts to transmit a packet. An attempt fails if a collision with a packet transmitted by another node occurs within the first 64 byte times of the transmission attempt. If a collision occurs the Ethernet interface waits for a random time between 0 and 2^k slot times where a slot time is defined to be equal to 64 byte times and $k = min(\textit{attempt number}, 10)$. If 16 attempts to transmit the packet fail the Ethernet interface discards the packet and generates an error event with an `error_code` of `ERROR_TX_PACKET_FAILED`. The first 64 (`FAILED_PACKET_LENGTH`) bytes of the packet are returned. See `B431_Wait-for_Event()`.

This mechanism is known as truncated binary exponential backoff.

The retry algorithm can be disabled by initialising the Ethernet interface with the `DISABLE_TX_RETRY` mode flag set. This will cause the Ethernet interface to immediately discard packets if it fails on the first transmission attempt.

5.1.5 Ethernet statistics

The device driver accumulates statistics that correspond to various operating characteristics of the Ethernet interface. They may be requested at any time, by calling `B431_Ether_Stats()`, and are returned to the application program via `B431_Waitfor_Event()`. ANSI C programmers will receive the statistics in a structure of type `ETHER_STATS`, occam programmers will receive an `INT32` array (appropriate indices are declared in `b431io.inc`).

With the exception of the time domain reflectometer value, which is described in the next section, all statistics are unsigned long integers which monotonically increase from 0.

The statistics are implicitly cleared to 0 when the Ethernet interface is initialised or may be explicitly cleared by calling `B431_Reset_Stats()` . The following statistics are accumulated:–

`tx_packets`	The total number of packets queued for transmission.
`rx_packets`	The total number of packets received without error.
`framing_errors`	The total number of packets received with a framing error, the packet contained a non–integer number of octets and also failed the CRC check.
`crc_errors`	The total number of packets received that failed the CRC check.
`packets_dropped`	The total number of packets dropped (discarded) because the inbound packet queue ran out of buffers during reception, buffers are released when packets are sent to the application program. See `B431_Waitfor_Event()`, this usually indicates a failure of application software to match the receive packet rate.
`packets_missed`	The total number of packets missed completely because the inbound packet queue was full, see above.
`deferred_transmits`	The number of occurrences of deferal during transmission, provides a measure of Ethernet loading. The ratio of this value and `tx_packets` can be regarded as an Ethernet utilisation factor.

`late_collisions`	The number of late collisions detected. A transmit packet collision was detected after the transmit slot time elapsed, the Ethernet interface does not retry after a late collision, the packet is discarded. This indicates a potential network fault as all nodes should obey the physical layer CSMA/CD access protocol.
`carrier_lost`	The number of occurrences of carrier loss during transmission, the Ethernet interface will continue to transmit the packet and will not retry. Indicates a potential network fault.
`no_more_retries`	The number of packets discarded because the Ethernet interface failed to transmit them after 16 retry attempts. Indicates a potential network fault, if the ratio of **`tx_packets`** to this value is high then suspect a cable fault. The time domain reflectometer value can provide an estimate of the distance to a potential cable fault, see the next section for more details.
`single_retries`	The total number of packets successfully transmitted after a single retry attempt, provides a measure of network loading.
`multiple_retries`	The total number of packets successfully transmitted after multiple retry attempts, also provides a measure of network loading.

5.1.6 Time domain reflectometer

The Ethernet interface has a hardware counter called the time domain reflectometer (TDR). When a packet is transmitted the counter starts counting at the transmit bit rate (10MHz), it stops counting when a packet collision is detected. If repeated retry failures occur then this value can provide an estimate of the distance to the location of a potential cable fault. The value returned (with other statistics) by calling **`B431_Ether_Stats()`** is an average of the time domain reflectometer for each occurence of transmit retry failure.

If the propagation velocity of the cable is P, then the distance (in meters) to a potential cable fault is given by:–

$$distance = \frac{1}{2} \times P \times TDR \times 0.1 \times 10^{-6}$$

The factor of $\frac{1}{2}$ results from the TDR being equal to the time for the transmission signal to propagate to the collision point and for the collision signal to return.

5.1.7 Heartbeat monitor

Heartbeat, otherwise known as Signal Quality Error, is a feature sometimes provided by Media Attachment Units for continuously verifying correct operation of the collision detection circuitry and associated cabling. This function is usually a jumper configured option within the MAU box.

Heartbeat works as follows: immediately after every transmission the MAU will assert it's collision detect signal. The Ethernet interface will correspondingly check that this has happened and if it fails to detect the collision (heartbeat) signal an error will be reported. `B431_Waitfor_Event()` will complete with an `error_code` of `ERROR_HEARTBEAT_STOPPED`.

The Ethernet interface should be initialised with the heartbeat monitor function enabled if the MAU supports it.

5.1.8 Performance

The following graph depicts data rate versus packet length. Results were obtained using a program designed to continuously transmit packets, with a constant packet length, to a non-existent destination address. The time taken to transmit a large number of packets was measured under fairly light Ethernet load conditions and the effective data rate calculated.

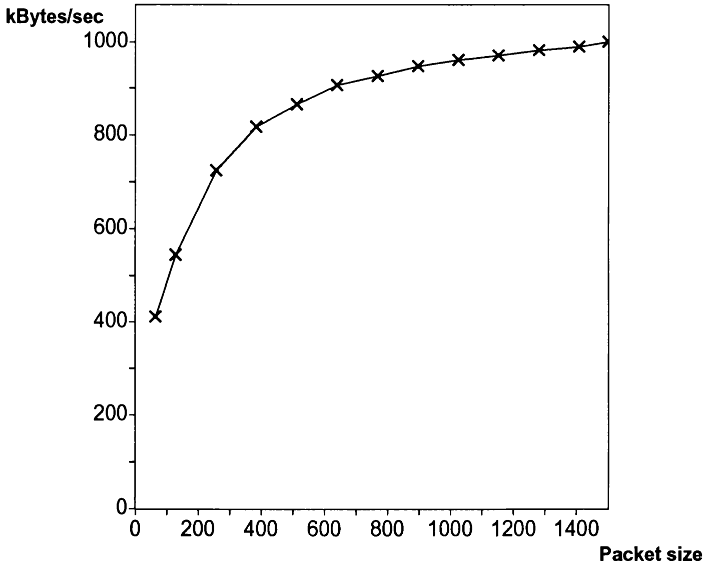

Figure 5.2 Graph showing packet size vs data rate

6 Detailed hardware description

6.1 Data structures

The Ethernet interface on the IMS B431 is implemented with the AM 7990 (LANCE). The LANCE avoids loading the IMS T222 with frequent I/O operations by having direct memory access (DMA). The LANCE transmits Ethernet packets directly from a set of transmit buffers, and receives packets directly into a set of receive buffers. The programmer can place the transmit and receive buffers in any convenient areas of memory.

Each buffer has an associated descriptor. The descriptors arbitrate buffer ownership between IMS T222 and LANCE and provide comprehensive error and status reporting for each packet received or transmitted. Each descriptor points to a single buffer. Refer to figure 6.1.

The descriptors are arranged in two blocks of memory called the *transmit message descriptor ring* and the *receive message descriptor ring*. The LANCE has a pointer to each of these rings. Each ring entry must start on a quadword boundary, i.e. at a byte address divisible by eight. The ring entries and their respective buffers are used strictly in the order in which they appear in the ring, entries are never skipped.

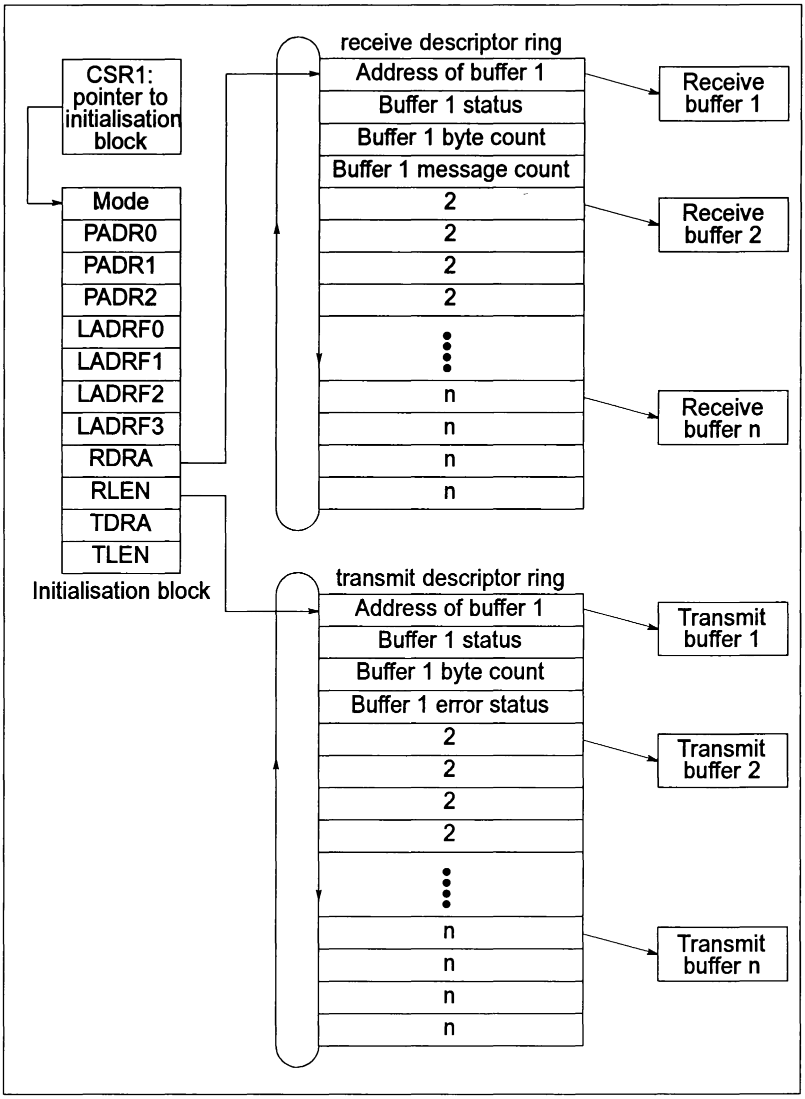

Figure 6.1 Data structures used by the LANCE

6.1.1 Buffer and descriptor ownership

The host and LANCE communicate with each other by means of the receive and transmit message descriptor rings. Ring entries (descriptors) and their associated buffers are *owned* either by the host or by the LANCE. Ownership is indicated by an ownership bit in each message decriptor. The LANCE does not access descriptors and buffers which it does not own, the host software should not access descriptors and buffers which it does not own. In this way, only valid data is exchanged between LANCE and host.

6.1.2 Data chaining

To make efficient use of the system memory, the LANCE incorporates a *data chaining* mechanism. This allows the transmit and receive buffers to be smaller than the maximum packet size (1514 bytes). Packets longer than the buffer size are written to several buffers and the buffers chained by setting appropriate bits in their descriptors. In this way, small packets occupy single small buffers and large packets occupy more buffers, so that little memory space is wasted. Data chaining can operate for both transmitted and received packets. The minimum packet or buffer size for data chaining is 100 bytes.

6.2 Software structure

The transputer's hardware support for multiple processes running in parallel makes it ideal as a communications controller. In a typical application, software running on the IMS T222 inputs packets to be transmitted on one (or more) of its links and outputs received packets on one (or more) of its links. The software would be structured as shown in figure 6.2.

The transmit queue monitor inputs packets for transmission from a channel (which may be a link and places them in the transmit buffers). It also monitors the progress of all packets queued for transmission. Any change in the status of a packet queued for transmission is signalled by a transmitter interrupt which the event handler passes on to the transmit queue monitor.

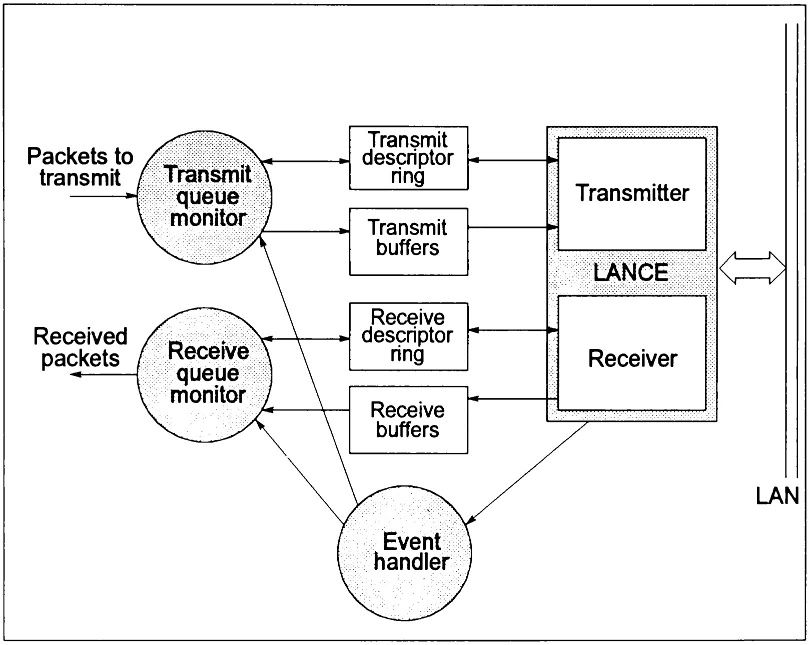

Figure 6.2 Software structure and interaction with hardware

The receive queue monitor waits for receiver interrupts from the event handler. It then examines the receive descriptor ring to determine the cause of the interrupt. It outputs correctly received packets on a receive channel.

The event handler determines whether interrupts were generated by the transmitter or receiver and passes them on to the transmit and receive handlers accordingly.

To transmit a packet, the IMS T222 simply has to place the packet in an empty transmit buffer and set up the descriptor for that buffer to indicate that it should be transmitted. The LANCE will transmit the buffer contents when it has transmitted all packets ahead of it in the descriptor ring. It then updates the descriptor contents to inform the IMS T222 of the packet status. It may also interrupt the IMS T222.

When the LANCE receives a packet from the Ethernet (or Cheapernet) it places the packet in the next available empty receive buffer. It then updates the descriptor for that receive buffer to indicate that it contains a received packet. It may also interrupt the IMS T222. When the IMS T222 has removed the received packet from the buffer it marks the descriptor as empty again.

Operation of the IMS B431 is the same regardless of whether it is connected to an Ethernet system or a Cheapernet system: the same software can be used with both types of network.

6.3 Initialising

Initialising the IMS B431 consists of setting up the physical and logical addresses of the node, and the data structures required for transmitting and receiving packets.

The LANCE needs to know the node addresses, and the location of and number of entries in the transmit and receive descriptor rings. The host sets these via the *initialisation block*. This is a parameter block placed at a defined address in memory. The LANCE initialises itself with these parameters on receipt of a command from the host.

The LANCE contains four control and status registers: CSR0-3. CSR1 and CSR2 are a pointer to the initialisation block. This must be written by the host before giving the initialise command. Setting the INIT bit in CSR0 causes the LANCE to load itself with the parameters from the initialisation block pointed to by CSR1 and CSR2. CSR3 defines the way in which the LANCE's DMA interface works.

6.3.1 The initialisation block

The initialisation block defines the logical and physical addresses of this node and the LANCE operating mode. It also defines the number of entries in and location of the transmit and receive message descriptor rings. The initialisation block is read by the LANCE when the host sets the INIT bit in CSR0. The structure of the initialisation block is shown in figure 6.3; it occupies twelve words of contiguous memory starting on a word boundary.

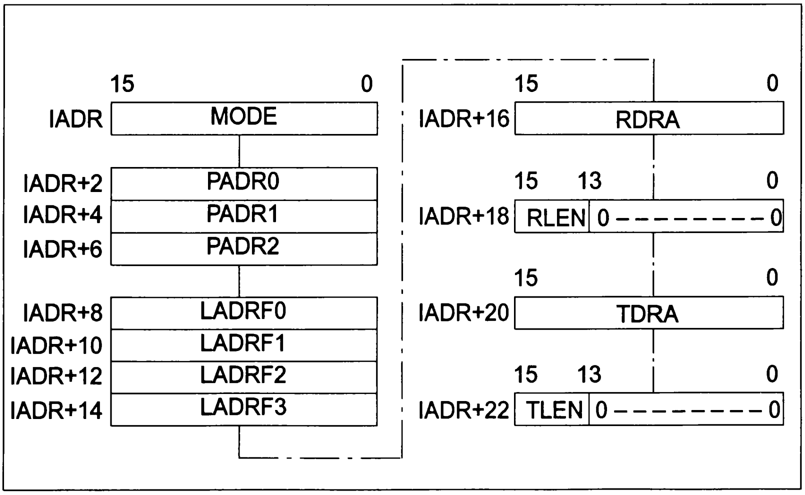

Figure 6.3 Initialisation block

Mode register

The MODE register is normally set to #0000. Non-zero values are used for test purposes, see section 6.13.

Physical Address (PADR0–PADR2)

These three words form the 48 bit address allocated to this node. PADR0 is the least significant part of the address. This address should be unique on the network and, in theory, can be unique for every ethernet node in existence. Physical addresses are always even. When the LANCE detects the start of a packet on the LAN, it examines the packet's destination address. If the destination address is #FFFFFFFFFFFF the packet is received as this is the broadcast address. If the least significant bit of the destination address is 0, it is compared with the physical address; if they are the same, the packet is received. Otherwise, the address is a logical address.

Logical Address Filter (LADRF0–LADRF3)

Logical addressing allows nodes with different physical addresses to receive the same packet. If the least significant bit of the destination address is 1, but the address is not the broadcast address (#FFFFFFFFFFFF), the destination address is a logical address. The six high order bits of the CRC of the destination logical address are then used to select one bit of the logical address filter If that bit is 1 the packet is received. Thus, to receive packets with a particular logical address, a particular bit (determined by the CRC algorithm) must be set in the logical address filter. An algorithm for determining the bit which must be set is described in section 5.1.2.

Receive Descriptor Ring Pointer (RDRA)

The host writes this location with the address of the start of the receive buffer descriptor ring. The three least significant bits of this address must be 0 because the start of each descriptor is aligned on a quadword boundary.

Receive Descriptor Ring Length (RLEN)

The number of entries in the descriptor ring must be a power of two between 1 and 128. The number of entries is expressed as a three bit number; e.g. 000 for 1 entry, 001 for 2 entries, 111 for 128 entries. The host writes this number to the three most significant bits of RLEN, all other bits must be 0. Table 6.1 shows the contents of RLEN for each possible ring size.

RLEN/TLEN	Ring Entries
#0000	1
#2000	2
#4000	4
#6000	8
#8000	16
#A000	32
#C000	64
#E000	128

Table 6.1 Transmit and Receive descriptor ring size

Transmit Descriptor Ring Pointer (TDRA)

The host writes this location with the address of the start of the transmit buffer descriptor ring. The three least significant bits of this address must be 0 because the start of each descriptor is aligned on a quadword boundary.

Transmit Descriptor Ring Length (TLEN)

The number of entries in the descriptor ring must be a power of two between 1 and 128. The number of entries is expressed as a three bit number; e.g. 000 for 1 entry, 011 for 8 entries, 111 for 128 entries. The host writes this number to the three most significant bits of TLEN, all other bits must be 0. Table 6.1 shows the contents of TLEN for each possible ring size.

6.3.2 CSR0–CSR3

The LANCE has four control and status registers, shown in figure 6.4, which contain status and error bits and also the control bits used to start, stop, and initialise the LANCE. CSR1 and CSR2 form a pointer to the initialisation block. CSR3 defines the way in which the DMA hardware operates.

The control and status registers are all accessed at the same location in the transputer's address space; `#7FFC`. The register to be accessed is identified by writing 0, 1, 2, or 3 to `#7FFE`; this location is known as the register address port (RAP).

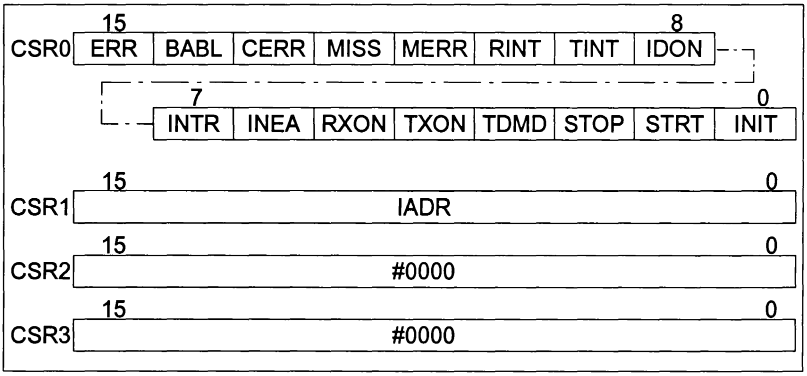

Figure 6.4 The control and status registers

CSR0

CSR0 contains the status and error bits. All of these bits can be read. Some of these bits are capable of causing interrupts when interrupts are enabled. All bits in CSR0 are cleared when the IMS B431 is reset except STOP which is set. Reset places the LANCE in STOP mode. All bits in CSR0 except STOP are cleared when STOP is set. The bits are

ERR

ERR = BABL + CERR + MISS + MERR

ERR is set by the LANCE and can be cleared only by clearing the condition causing the error.

BABL (Babble Error)

Babble Error is set by the LANCE after 1519 bytes have been transmitted: i.e. if the packet length is greater than the maximum allowed by the ethernet specification. The LANCE continues to transmit until the whole packet has been transmitted or an error occurs. BABL is cleared by writing 1 to this bit.

CERR

As a test of the collision detection logic, the LANCE expects its *collision detect* input to be asserted within 2μs after the end of each transmission. It is up to the MAU to assert the collision pair at the SIA connector. If it is not asserted or if there is a fault in the collision detection circuitry, the LANCE will set CERR. This feature is known as heartbeat. CERR is an indication of whether the collision detection logic is functional. If the MAU does not support heartbeat, CERR should be ignored. CERR does not set INTR.

MISS (Receiver Miss Error)

The LANCE sets MISS when it loses a packet because it did not own a receive buffer. MISS is cleared by writing 1 to this bit.

MERR

The LANCE sets MERR if it could not access the memory on the IMS B431. This condition should never occur as it indicates a hardware fault. MERR should never become set. MERR is cleared by writing 1 to this bit.

RINT

The LANCE sets Receiver Interrupt when it updates the Receive Descriptor Ring entry for the last buffer of a received packet: i.e. after a packet is received. RINT is cleared by writing 1 to this bit.

TINT

The LANCE sets Transmitter Interrupt when it updates the Transmit Descriptor Ring entry for the last buffer of a transmitted packet: i.e. after a packet is transmitted. TINT is cleared by writing 1 to this bit.

IDON

The LANCE sets Initialisation Done when it has completed it's self initialisation procedure started by setting the INIT bit. IDON is cleared by writing 1 to this bit.

INTR

INTR = BABL + MISS + MERR + RINT + TINT + IDON

If INEA is 1, the LANCE will assert the transputer's **Event** pin in conjunction with setting INTR.

INEA (Interrupt Enable)

If INEA is 1, the LANCE will assert the transputer's **Event** pin if INTR is set. If INEA is 0, the LANCE will not assert the **Event** pin.

RXON (Receiver On)

Indicates whether the LANCE receiver circuitry is enabled (1) or disabled (0).

TXON (Transmitter On)

Indicates whether the LANCE transmitter circuitry is enabled (1) or disabled (0).

TDMD (Transmit demand)

Setting TDMD causes the LANCE to access the transmit descriptor ring immediately instead of waiting for it's normal polling interval of 1.6ms to elapse. It can be used by the transmit driver software to speed up transmission. TDMD is cleared by the LANCE after is is recognised.

STOP

Setting STOP has the effect of resetting the LANCE. Transmission and reception cease and the LANCE remains idle until STRT or INIT is set. STOP clears all other bits in CSR0. STOP can only be cleared by setting STRT or INIT.

STRT

The driver software sets STRT to cause the LANCE to begin transmission and reception of packets. STRT can only be cleared by setting STOP.

INIT
The driver software sets INIT. This causes the LANCE to initialise itself with the parameters from the initialisation block. STOP must be set before INIT can be set.

Bit	Set By	Cleared By	Write 1	Write 0	Sets INTR
INIT	write 1	STOP	1	–	no
STRT	write 1	STOP	1	–	no
STOP	write 1	STRT/INIT	1	–	no
TDMD	write 1	LANCE	1	–	no
TXON	DTX=0	DTX=1/MERR/UFLO/BUFF	–	–	no
RXON	DRX=0	DRX=1/MERR	–	–	no
INEA	write 1	write 0	1	0	no
INTR	BABL+MISS+MERR+RINT+TINT+IDON		–	–	–
IDON	LANCE	write 1	0	–	yes
TINT	LANCE	write 1	0	–	yes
RINT	LANCE	write 1	0	–	yes
MERR	LANCE	write 1	0	–	yes
MISS	LANCE	write 1	0	–	yes
CERR	LANCE	write 1	0	–	no
BABL	LANCE	write 1	0	–	yes
ERR	BABL+CERR+MISS+MERR		–	–	no

Table 6.2 CSR0 bits

Notes:

1 The **Set By** and **Cleared By** columns describe the circumstances under which each bit is set and reset. In addition, all bits (except STOP) are cleared by STOP and reset: STOP is set by reset.

2 The **Write 1** and **Write 0** columns show the effect of writing each bit with 1 or 0 respectively. A dash in either of these two columns indicates that writing that value has no effect on that bit.

3 The **Sets INTR** column shows which bits will set the INTR bit if they themselves are set. These are the bits which can cause interrupts to the IMS T222 if interrupts are enabled (INEA=1).

4 INTR and ERR simply reflect the states of other bits in CSR0. They are cleared by clearing the bit(s) causing the interrupt or error.

The bits in CSR0 can be divided into two groups: the *control* bits, and the *status* bits. The control bits are **INIT, STRT, STOP, TDMD,** and **INEA.** The status bits are **TXON, RXON, INTR, IDON, TINT, RINT, MERR, MISS, CERR, BABL,** and **ERR.**

The way in which CSR0 works allows operations on it to be performed very easily. For example, to clear a transmitter interrupt and enable interrupts, simply write `#0240` - no other bits will be affected. Writing zeroes to the register has no effect on any bit except INEA, so interrupts can be disabled without affecting any other bit simply by writing `#0000`. Writing a one to most of the status bits will clear that bit: writing a one to a control bit will cause the LANCE to perform the required action. Table 6.2 lists the details.

CSR1

CSR1 holds the 16 bit byte address of the start of the initialisation block; note that the address must be even since the initialisation block must start on a word boundary.

CSR2

CSR2 extends the address held in CSR1 to a 24 bit value. Since the IMS B431 is a 16 bit addressed system, CSR2 is always written with `#0000`.

CSR3

CSR3 defines the way in which the LANCE's DMA interface works. CSR3 must be written with `#0000`; otherwise the IMS B431 will not function correctly.

6.3.3 Summary

Before any data packets are sent or received, the LANCE must be initialised and the correct data structures set up in memory. The initialisation procedure for normal transmission and reception is as follows.

1 set the STOP bit and disable interrupts by writing `#0004` to CSR0.

2 write the byte address of the initialisation block to CSR1; note that the address must be even.

3 write `#0000` to CSR2 and `#0000` to CSR3.

4 write TDRA with the start address of the transmit descriptor ring.

5 set TLEN to indicate the transmit descriptor ring length.

6 write RDRA with the start address of the receive descriptor ring.

7 set RLEN to indicate the receive descriptor ring length.

8 write a suitable value to the logical address filter; LADRF. This will be 0 if logical addressing is not being used.

9 write the node's physical address to the physical address register: PADR.

10 write `#0000` to the MODE register.

11 set up the transmit and receive message descriptors.

12 initialise the LANCE and enable interrupts by writing `#0041` to CSR0.

The LANCE then initialises itself by reading the initialisation block. At the end of initialisation the LANCE sets IDON in CSR0 and interrupts the host.

- The host writes `#0142` to CSR0 to clear the interrupt, enable interrupts, and start transmission and reception of packets.

The LANCE will then transmit packets from the transmit buffers as they are written by the host and will write received packets to the receive buffers as they are received.

6.4 Receiving

The LANCE attempts to receive all packets which are addressed to this node and all packets with the broadcast address. The LANCE communicates received packets to the host by means of the *receive message descriptor ring* and receive data buffers. The receive data buffers are accessed via the pointers in the receive descriptor ring entries. The receive descriptor ring is shown in figure 6.5, it consists of a four word descriptor (entry) for each receive buffer. Each entry must start on a quadword boundary (a byte address divisible by eight). The LANCE uses the receive message descriptors and buffers strictly in the order in which they appear in the descriptor ring: entries are never skipped.

Received packets will be *data chained* over several receive buffers if they are too long to fit in a single receive buffer. This is recorded in the message descriptor for each receive buffer used.

6.4.1 The receive message descriptor

Each receive descriptor ring entry consists of the four words RMD0-RMD3 shown in figure 6.5. Each entry points to a *receive buffer* and stores information about that buffer such as: whether it is owned by the LANCE or the host, its length, the number of message bytes received into that buffer, and whether any errors occurred during reception of the buffer contents.

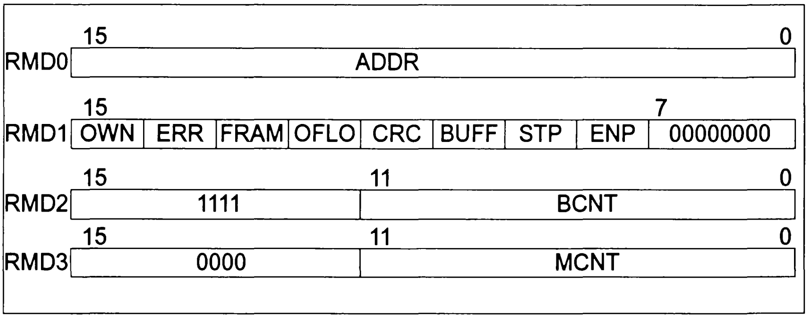

Figure 6.5 Receive message descriptor ring entry

Receive message descriptor 0 (RMD0)

RMD0 is a pointer to the receive data buffer referred to by this descriptor. On receipt of a packet the LANCE will write the received data to the block of memory at this address.

Receive message descriptor 1 (RMD1)

RMD1 is used to arbitrate buffer ownership between the LANCE and the host. It is also used by the LANCE to indicate whether any errors occurred during reception of this packet and to indicate whether this received message is continued into the next buffer (data chaining). The bits in RMD1 are

OWN

1 if this descriptor and buffer are owned by the LANCE.

0 if this descriptor and buffer are owned by the host.

The host software should not access the descriptor or buffer if they are owned by the LANCE as their contents will not be valid. The host software should set OWN to 1 to hand this descriptor and buffer back to the LANCE when it has finished with them: it should never set OWN to 0. The LANCE will set OWN to 0 when data has been received into the buffer or if an error occurred while data was being received into this buffer. The LANCE will never set OWN to 1.

ERR

ERR = FRAM + OFLO + CRC + BUFF

ERR is a summary of the other receive error bits.

FRAM

Framing Error. The LANCE sets FRAM if there was a CRC error and the packet contained a non-integer multiple of 8 bits. If there was no CRC error FRAM is not set, regardless of the number of bits in the packet. Thus, a FRAM error occurs if the packet has become truncated or extended in transit.

OFLO

Overflow Error. The LANCE sets OFLO if its internal buffer overflowed because no receive buffers were available. This would happen if packets are received faster than the host software empties the receive buffers.

CRC

CRC Error. The LANCE sets CRC if the CRC calculated on the received packet does not agree with the CRC appended to the packet. This indicates that the packet has been corrupted in transit.

BUFF

Buffer Error. The LANCE sets BUFF if it needs to data chain the received packet into the next buffer but cannot because it does not own the next receive buffer.

This would happen if packets are received faster than the host software empties the receive buffers.

STP

Start of Packet. The LANCE sets STP in the descriptor for the first buffer containing part of a data chained, received packet. The LANCE also sets STP if the packet is not data chained (it occupies a single receive buffer).

ENP

End of Packet. The LANCE sets ENP in the descriptor for the last buffer containing part of a data chained, received packet. The LANCE also sets ENP if the packet is not data chained (it occupies a single receive buffer).

Receive message descriptor 2 (RMD2)

RMD2 is the two's complement of the length of the buffer pointed to by this descriptor: i.e. it is *minus* the length of the buffer. It is the maximum number of bytes which the LANCE is allowed to write to this buffer. If a larger packet is received, the LANCE will continue the received data in the buffer pointed to by the next descriptor in the receive descriptor ring: i.e. it will data chain the packet.

Receive message descriptor 3 (RMD3)

The LANCE writes the length of the received packet into RMD3 of the last descriptor used by this packet. Valid only when ERR is clear and ENP is set: i.e. in the last descriptor of a correctly received packet. Thus, for non data-chained packets, RMD3 contains the number of message bytes in the buffer. For data chained packets, the length of the message is held in RMD3 in the descriptor for the last buffer used.

6.5 Size and number of receive buffers

If the packet size is fixed or it is known that the majority of received packets will be of a certain size; then it is most efficient to have receive data buffers of that size.

When the size of received packets is unknown, the optimum receive buffer size depends on the trade off between several factors. The size of received packets can be up to 1518 bytes. It is inefficient to use receive buffers of this size because this results in wasting large amounts of memory when small packets are received. However, the LANCE will chain received packets which are larger than the buffer size over several buffers. This allows buffers which are smaller than the maximum packet size to be used and so uses the memory more efficiently. The overhead in the chaining operation is small but not negligible since it requires more work from the receive driver software, the minimum buffer size which allows data chaining is 100 bytes.

The number of buffers and descriptor ring entries must be a power of 2. There should be as many receive buffers as possible within the limits of the memory space available: the number of bytes used by each buffer is the length of the buffer plus 8 bytes for the descriptor.

6.6 LANCE actions during packet reception

When no packets have been received, or when the host has dealt with all received packets, all of the receive message descriptors and buffers are owned by the LANCE: i.e. OWN=1 in all buffers. Thus, the initialisation software should set the OWN bits in all the receive buffer descriptors.

The LANCE maintains a pointer to the next descriptor it will use. This points to descriptor 0 after initialisation and, after each packet is received, it points to the descriptor after the last one used by that packet. After the last descriptor in the ring is used, the next to be used is descriptor 0. Thus, the LANCE uses the descriptors as a ring and never skips descriptors.

When a packet arrives, the LANCE checks that it owns the receive descriptor which it points to. If it does, it writes the received data to the buffer pointed to by this descriptor, and updates the descriptor. It marks the descriptor as being owned by the host and interrupts the host to inform it that a packet has been received. If it does not own the descriptor which it points to, it cannot receive the packet and sets the miss error bit (MISS) in CSR0. This action also sets ERR and INTR in CSR0 and, if interrupts are enabled, interrupts the IMS T222. A higher level protocol may deal with the missed packet by sending a negative acknowledge packet or requesting retransmission when it has receive buffers available.

The LANCE reads the buffer length from RMD2. If the received packet is longer than the receive buffer, the LANCE attempts to chain it into the next buffer. The LANCE examines the next descriptor to determine whether it owns the buffer. If it does not, it cannot continue to receive the packet. In this case, LANCE sets BUFF and ERR in RMD1 of the current descriptor (not the one it does not own) and sets RINT and INTR in CSR0. If interrupts are enabled, this action will interrupt the IMS T222. If the LANCE does own the next buffer, it will continue to receive into that buffer. Data chaining will continue until either the packet ends, or the next buffer is not owned by the LANCE.

When the whole packet has been received, the LANCE sets RINT and INTR in CSR0. If interrupts are enabled (INEA=1) it will also interrupt the IMS T222.

6.7 Receive driver actions

When the entire packet has been received, or if an error occurs during reception, the LANCE interrupts the host. The host can determine that the interrupt was caused by reception of a packet (regardless of whether there was an error) by examining RINT in CSR0. RINT will be set if the interrupt was caused by a receive event.

The host software should maintain a pointer (RXDESR) to the receive descriptor for the next buffer it expects to have to process. The initialisation software must set RXDESR to point to descriptor 0 as this will be the first used by the LANCE. It should also maintain a pointer to the descriptor which it is currently processing (*current.rx.descriptor.ptr*). When a receive event causes the receive driver to be entered, this pointer should be a copy of RXDESR.

The host software should examine RMD1 of the descriptor pointed to by *current.rx.descriptor.ptr*. OWN should be clear indicating that the host now owns this descriptor and its associated buffer. If OWN is set the descriptor is not valid. The host software's first objective is to find the last descriptor containing part of the received packet. The last used descriptor will have either ENP, ERR or both set. Thus, the host software should examine the descriptors until it finds one with ERR or ENP set. It should examine the OWN bit in each descriptor before examining the other bits. It should find OWN clear. If it finds OWN set, the descriptor is not valid.

An invalid descriptor indicates that the host and LANCE are out of step or some serious error has occurred. The only sensible method of recovery is to stop and reinitialise the LANCE.

6.7.1 ERR is 0

No errors occurred while this buffer was being received. The host should check for ENP set in this descriptor. If ENP is set, then this is the last buffer containing this packet. If ENP is not set, the host should examine RMD1 of the next descriptor in the ring. Again OWN should be 0 and ERR and ENP should be checked. The host should continue examining descriptors until it either finds one with ERR=1 or with ENP=1.

6.7.2 ERR is 1

ERR is set if an error occurred while data was being received into this buffer If ERR is set, this is the last descriptor/buffer used by this packet. Four types of error can occur, errors are reported only in the last descriptor used by a packet.

- There was a CRC error; CRC is set. A CRC error is only detected at the end of a packet.
- The packet did not contain a multiple of eight bits and there was a CRC error; FRAM is set. A framing error is only detected at the end of a packet.
- The LANCE's internal buffer overflowed; OFLO is set. This occurs when the LANCE needed to data chain into the next receive buffer, but the next receive buffer was owned by the host.
- The received packet is longer than the current receive buffer and the next buffer is OWNed by the host; BUFF is set. OFLO will also be set if this caused the LANCE's internal buffer to overflow.

CRC and FRAM (framing) errors indicate that the received packet was corrupted and should be discarded. OFLO and BUFF errors indicate that all or part of the packet had to be discarded. A higher level protocol may be able to recover by causing the packet to be retransmitted.

The difference between BUFF and OFLO is that BUFF is set any time the LANCE needs to data chain but cannot because the next buffer is owned by the host. OFLO is set if more data is received after a BUFF error causing the LANCE's internal buffer to overflow. Thus, the LANCE may set BUFF and OFLO together but OFLO should never appear set on its own.

If an error occurred, the receive driver should hand all the descriptors and buffers back to the LANCE: by writing #8000 to RMD1 and #0000 to RMD3. RXDESR should be set to point to the descriptor after the one which had ERR set.

6.7.3 ENP is set (and ERR is clear)

No errors occurred during reception of the packet. The packet has been correctly received. *current.rx.descriptor.ptr* now points to the last descriptor used by this packet.

The receive driver reads the packet length from RMD3 and determines the number of bytes in the last used buffer. It should then remove the correctly received packet from the receive buffers, starting with the data from the buffer pointed to by RXDESR and ending with the data from the buffer pointed to by *current.rx.descriptor.ptr*. It should then hand these buffers back to the LANCE for reuse by

- writing #8000 to RMD1 in each descriptor,
- writing #0000 to RMD3 in each descriptor,
- RMD0 and RMD2 may be changed.

This marks them as owned by the LANCE and clears the message count in each descriptor. Changing RMD0 implies that the buffer containing the received data is detached from the descriptor ring and another (empty) buffer is attached in its place. This may be a useful thing to do since it means that the receive buffer is freed quickly for reuse by the LANCE.

RXDESW is then updated to point to the descriptor after that pointed to by *current.rx.descriptor.pointer*. This will be the next to be used by the LANCE for received data. If *current.rx.descriptor.ptr* points to the last descriptor in the ring, RXDESW should be set to point to descriptor 0.

6.8 Transmitting

The host queues packets for transmission by the LANCE by means of the *transmit message descriptor ring*. The ring is shown in figure 6.6, it consists of a four word transmit descriptor (entry) for each transmit buffer. The LANCE uses the transmit message descriptors and buffers strictly in the order in which they appear in the descriptor ring: entries are never skipped.

The host software can *data chain* packets for transmission over several transmit buffers if they are too long to fit in a single transmit buffer. The host indicates this to the LANCE in the message descriptor for each receive buffer used.

6.9 The Transmit message descriptor

Each transmit descriptor ring entry consists of the four words TMD0-TMD3 shown in figure 6.6. Each entry points to a *transmit buffer* and stores information about that buffer such as: whether it is owned by the LANCE or the host, its length, and whether any errors occurred during an attempt to transmit the buffer contents.

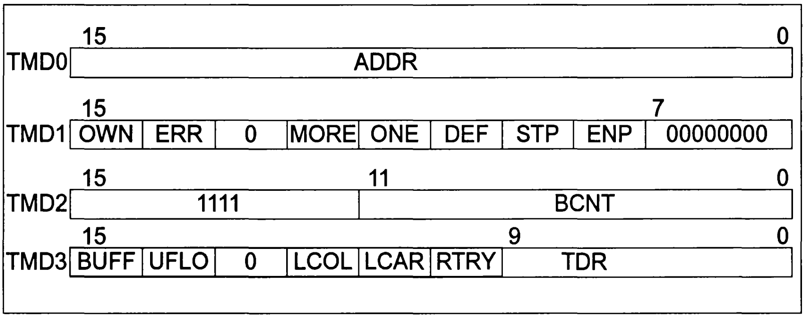

Figure 6.6 Transmit message descriptor ring entry

6.9.1 Transmit message descriptor 0 (TMD0)

TMD0 is a pointer to the start of the block of transmit data referred to by this descriptor. This is a byte address, transmit buffers can start at any byte address.

6.9.2 Transmit message descriptor 1 (TMD1)

TMD1 is used to arbitrate buffer ownership between the LANCE and the host. It is also used by the LANCE to indicate whether an error prevented transmission of the packet and, if transmission was successful, whether any retries were necessary. The bits in TMD1 are:

OWN

OWN is set by the transmit driver software to indicate that this descriptor is valid and points to a buffer containing valid data for transmission. When the LANCE next comes to examine this descriptor (having dealt with all previous descriptors in the transmit ring) it will attempt to transmit the contents of this buffer. OWN is cleared by the LANCE after it has transmitted the packet or has failed to transmit the packet because of errors. The LANCE never sets OWN, the host software should never clear it. If the LANCE finds OWN clear it will not access the descriptor further and will not access the buffer. The host should not access a descriptor or buffer when it finds the OWN bit set.

ERR

ERR is set by the LANCE if an error occurred during an attempt to transmit this buffer.
ERR = LCOL + LCAR + UFLO + RTRY
See section 6.9.4.

MORE

Set by the LANCE if more than one retry was needed to transmit the packet. Useful as an indication of network loading.

ONE

Set by the LANCE if exactly one retry was needed to transmit the packet. Useful as an indication of network loading.

DEF

Set by the LANCE if it had to defer when attempting to transmit this packet. Useful as an indication of network loading.

STP

The transmit driver software should set STP to indicate that this descriptor points to the first buffer containing part of a packet which is chained over several buffers. It should also be set when the packet occupies only a single buffer.

ENP

The transmit driver software should set ENP to indicate that this descriptor points to the last buffer containing part of a packet which is chained over several buffers. It should also be set when the packet occupies only a single buffer.

6.9.3 Transmit message descriptor 2 (TMD2)

TMD2 is the two's complement of the number of bytes in this buffer to be transmitted; i.e. it is *minus* the number of bytes to be transmitted.

6.9.4 Transmit message descriptor 3 (TMD3)

TMD3 reports errors which have occurred during attempts to transmit this packet. It should be written to #0000 by the host before handing over to the LANCE. If any of the upper six bits are set when the buffer is handed back to the host, the LANCE has failed to transmit the packet.

BUFF (buffer error)

Set by the LANCE during transmission if it did not find ENP set in this descriptor and did not OWN the next buffer: i.e. if it did not find the last buffer containing part of the packet being transmitted. Transmit driver software can avoid buffer errors by not setting the OWN bit in the first descriptor for a particular packet until all buffers and descriptors for that packet have been set up.

UFLO (underflow error)

The LANCE sets underflow error when the system memory does not respond to an attempt by the LANCE to read transmit data. It should never occur on the IMS B431.

LCOL (late collision)

The LANCE sets LCOL if a collision occurred after the first 64 bytes of the packet were transmitted. This indicates a fault in the network such as a network which is larger than the maximum size allowed by IEEE802.3 or a faulty transmitter elsewhere on the network.

LCAR (loss of carrier)

The LANCE sets LCAR if it cannot detect its own transmission. The LANCE monitors its own transmissions as a test of the transmit and receive paths to the ethernet cable and back. If LCAR is set this indicates faults such as a faulty or disconnected transceiver, fault on the ethernet cable, faulty serial interface adapter. Not valid in internal loopback test mode.

RTRY (retry error)

The LANCE sets RTRY if repeated collisions caused 16 attempts to transmit the packet to fail. Transmission retries can be disabled by setting DRTY in the mode register: RTRY will then be set after one failed transmission attempt. The LANCE will not attempt to transmit the packet again if it has set RTRY. A higher level protocol may reschedule the packet for transmission.

TDR (collision timer)

Valid only if RTRY is set. The value in TDR is the time (in units of 0.1ms) from the start of transmission until the collision was detected. If the propagation velocity of the cable is known, it can be used to determine the approximate distance from this node to the point where the collision occurred. Repeated collisions at the same distance indicate a possible cable fault.

6.10 LANCE actions during transmission

When no packets are to be transmitted, or when the host has dealt with all received packets, all of the receive message descriptors and buffers are owned by the host: i.e. OWN=0 in all buffers. Thus, the initialisation software should clear the OWN bits in all the receive buffer descriptors.

The LANCE maintains a pointer to the next descriptor it expects to contain transmit data. This points to descriptor 0 after initialisation and, after each packet is transmitted, it points to the descriptor after the last one used by that packet. After the last descriptor in the ring is used, the next to be used is descriptor 0. Thus, the LANCE uses the descriptors as a ring.

The LANCE polls the descriptor, referenced by its pointer, every 1.6ms. When it finds the OWN bit set in this descriptor, it regards this as a signal that the buffer referred to by this descriptor contains valid data for transmission.

The LANCE then examines the STP and ENP bits in this descriptor. It should find STP=1 indicating the start of a packet. If STP=0, this descriptor is invalid, so the LANCE simply clears the OWN bit to hand the buffer back to the host. If STP=1, the LANCE reads the number of bytes in this buffer to be transmitted from TMD2 and starts to transmit the packet. If ENP is also 1, this buffer contains the whole packet.

If ENP=0, the packet is assumed to continue in the next buffer. The LANCE then examines the next descriptor. It should find OWN=1. If OWN=0 the LANCE sets BUFF in TMD3 as it cannot access the next part of the packet. It then sets TINT and INTR in CSR0 and interrupts the IMS T222 if interrupts are enabled. If OWN=1, the LANCE will transmit the contents of this buffer, without interruption, when it has finished transmitting the previous one. The contents of buffers are transmitted until either: a descriptor is encountered with ENP=1, or a descriptor is encountered with OWN=0. The buffer for which ENP=1 is transmitted.

Note that the host should write TMD2 in each descriptor with the number of bytes which are to be transmitted from this buffer, not the length of the packet.

Once the packet has been transmitted successfully, the LANCE updates the transmit message descriptor for the last buffer (the one with ENP=1), clears OWN in the descriptors used by this packet, and sets TINT and INTR in CSR0.

6.10.1 Failure to transmit

The LANCE may fail to transmit a data chained packet before reaching the last descriptor used by the packet. In this case, the errors are recorded in the descriptor for the buffer which was being transmitted when they occurred. The descriptors and buffers for the packet, up to and including the one in which the error occurred, are handed back to the host by setting OWN=0. The LANCE will then hand back the descriptors for the rest of the packet because, when it polls them, it will find that although it owns them, they do not have STP set. It will not attempt to transmit the rest of the packet.

Thus, all the descriptors and buffers for a packet which fails to transmit are handed back to the host. The LANCE will then poll the descriptor after the last used by the failed packet and, if this descriptor is valid, it will attempt to transmit the associated buffer.

6.11 A typical transmit driver

The host needs to maintain two pointers to entries in the transmit descriptor ring. One of these (TXDESR) points to the descriptor for the first buffer which has been queued for transmission but which has not yet been transmitted: i.e. the one which the LANCE will attempt to transmit next. The other (TXDESW) points to the descriptor after the last one which has been queued for transmission: i.e. it points to the next descriptor the host can use. Both of these pointers are initialised to point to transmit descriptor 0.

Transmitting a data packet consists of a few simple steps.

1 The host adds the destination address, source address, and length fields to the head of the packet.

2 The host checks that TXDESW points to a descriptor which is owned by the host.

3 The host writes the packet to the transmit buffer referred to by TXDESW. Large packets can be chained over several buffers.

4 The number of message bytes in each buffer, BCNT, is written to the descriptor for each transmit buffer occupied by the packet.

5 The *start of packet* bit, STP, is set in the descriptor for the first transmit buffer occupied by the packet.

6 The *end of packet* bit, ENP, is set in the descriptor for the last transmit buffer occupied by the packet.

7 The buffers occupied by this message are handed to the LANCE by setting the OWN bit in the descriptor for each buffer used. OWN should be set last in the descriptor for the first buffer used by this packet.

8 TXDESW is adjusted to point to the next free entry in the transmit descriptor ring; i.e. the one after the last entry used by this packet.

After each buffer is transmitted, the LANCE clears the OWN bit in the descriptor for that buffer. This indicates to the host that the LANCE has attempted to transmit the buffer. Several types of error may occur during transmission of a packet. If any error occurs, it is marked in the message descriptor of the buffer which was being transmitted at that time. The LANCE also sets *transmit interrupt* (TINT) in CSR0 and interrupts the host. If no errors occur, the LANCE sets TINT after the last buffer of the current message has been transmitted. It is up to the host to determine the nature of the interrupt by examining CSR0 and the transmit buffer descriptors.

On receiving an interrupt, the host software should always check TINT to see if it was caused by an attempt to transmit a packet. If TINT is set, the host software should examine the descriptor pointed to by TXDESR. It should find the OWN bit in that descriptor to be 0 indicating that the LANCE has attempted to transmit that buffer. If OWN=1, the LANCE and host are out of step: this must be treated as a fatal error and the LANCE must be re-initialised.

The host software's first task is to find the last descriptor belonging to this packet. This is done by by examining each descriptor, from the one pointed to by TXDESR onwards, until one is found with ENP=1. While it does this, it should also examine the BUFF and ERR bits in each descriptor as a check for transmit errors.

6.11.1 BUFF and ERR are clear

BUFF and ERR will be clear in all descriptors used by this packet if the packet was transmitted successfully. The LANCE will have handed all buffers and descriptor used by this packet back to the host.

6.11.2 BUFF is set

The LANCE sets BUFF if it did not find ENP set in this descriptor and did not find OWN set in the next descriptor; i.e if it expected to transmit the next buffer as part of this packet but did not own the next buffer. The LANCE completes transmission of the current buffer before setting BUFF and interrupting the host. The host can avoid BUFF errors by never setting the OWN and STP bits in the first descriptor for a packet until it has set up *all* the descriptors and buffers for the whole packet.

6.11.3 ERR is set

The LANCE sets ERR if:

- 16 attempts to transmit the packet have failed because of repeated collisions; RTRY is set,
- a collision occurred after the first 64 bytes of the packet were transmitted (presumably due to a faulty transmitter elsewhere in the network or a network which is too large); LCOL is set,
- the LANCE could not detect its own transmission (presumably due to a fault on the cable); LCAR is set,
- transmission of the packet was not completed because the LANCE was unable to read part of the packet from memory in time; UFLO is set.

In all of these cases, the transmission has failed and the host must act accordingly. The transmission may fail before all of the packet has been transmitted. This should cause a CRC error at the receiving node so that the receiving node will discard the partly transmitted packet.

Whether the error occurred during the last buffer of the packet or a previous one, the LANCE will hand all the buffers for a failed packet back to the host. TXDESR should now point to the descriptor after the last one used by this packet.

Example: transmitting a short packet

A short packet is one which is shorter than the transmit buffer size and, therefore, only occupies a single transmit buffer. Note that there is a minimum packet size of 64 bytes; this limit is imposed by the Ethernet specification. After adding the header information to the packet, the host sets up the transmit descriptor as illustrated in figure 6.7. TMD0 points to the start of the buffer containing the packet. TMD2 contains *minus* the length of the packet. TMD3 is cleared (all 0). TMD1 is written with `#8300`. This sets the STP and ENP bits indicating that this is the first and last buffer containing this packet. It also sets the OWN bit to indicate to the LANCE that this buffer should be transmitted.

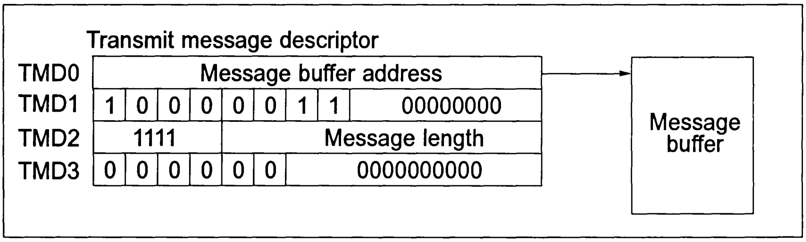

Figure 6.7 Transmitting a single data buffer

When the LANCE has transmitted the buffers for all previous descriptors in the ring, it examines the OWN and STP bits in this descriptor. Finding them set, it attempts to transmit the buffer. If the packet is transmitted successfully, the LANCE clears the OWN bit to give the descriptor and buffer back to the host. Then, it sets TINT and interrupts the host. If a fault or repeated collisions prevent transmission of the packet, the LANCE updates TMD1 and TMD3 accordingly and then sets TINT and interrupts the host.

Example: transmitting a long packet

A long packet is one which is longer than the transmit buffer size. Note that the Ethernet specification sets a maximum packet length of 1518 bytes. This means a maximum of 1500 data bytes as the 14 byte header and 4 byte CRC must be added to this. Packets longer than the transmit buffer size must be *data chained* over several transmit buffers; the first buffer used must contain at least 100 bytes. *Data chaining* is done by setting the STP bit in the descriptor for the first buffer only. The ENP bit is set in the descriptor for the last buffer only. Descriptors for other buffers have neither bit set. Once the host has arranged the packet to be transmitted in the buffers and has set the STP and ENP bits in the appropriate descriptors, it sets the OWN bit in the descriptor for each buffer used. The typical arrangement is illustrated in figure 6.8.

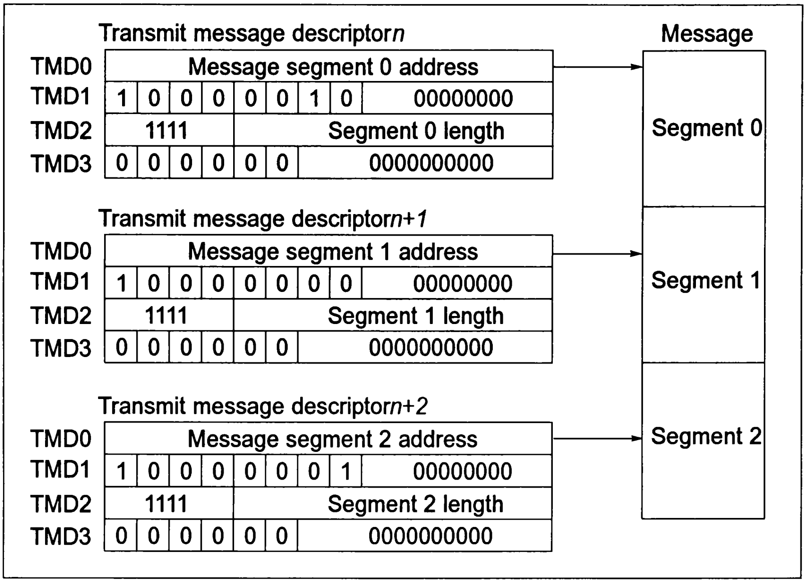

Figure 6.9 Transmitting a packet longer than the buffer size

When the LANCE has transmitted the buffers for all previous descriptors in the ring, it examines the OWN and STP bits in descriptor *n*. Finding them set, it attempts to transmit the buffer. Since ENP is not set, the LANCE examines the OWN bit in descriptor *n+1*. If OWN is not set, the LANCE transmits the current buffer and sets the BUFF bit in its descriptor to indicate the error to the host. If OWN is set the LANCE transmits this buffer also as part of the same packet. The LANCE continues to do this until it has transmitted the last buffer of this packet; i.e. the buffer for which ENP was set.

6.12 Interrupts, errors and error handling

The LANCE interrupts the IMS T222 for a variety of reasons. These are

- initialisation done
- successful reception of a packet
- successful transmission of a packet
- errors during transmission
- errors during reception.

6.12.1 Interrupt handling

The device driver software will normally incorporate an interrupt handler This must respond to interrupts in the order in which they occur It must also sort the interrupts and pass transmitter originated interrupts to the transmitter monitor and receiver originated interrupts to the receiver monitor. Some types of interrupt are fatal and require the LANCE and driver software to be restarted.

On receiving an interrupt, the device driver's first action is always to disable further interrupts by writing `#0000` to CSR0. CSR0 is then examined to determine the cause of the interrupt.

To avoid missing interrupts, and to preserve the time sequence of interrupt causing events the following method is suggested.

- interrupts are disabled by writing `#0000` to CSR0
- CSR0 is read and added to a list of values read from CSR0
- the value read is written back to CSR0 to clear the interrupt
- interrupts are enabled by writing `#0040` to CSR0.

Thus, each event which caused an interrupt is recorded by an entry in the list of values read from CSR0. The host deals with the list entries in the order in which they occurred and no events are missed.

Such an interrupt handler is best implemented as a *process* running in parallel with the receive and transmit drivers.

6.12.2 Errors

Errors can occur during both transmission and reception of packets. Most errors are recorded in the transmit or receive message descriptors for the buffers which were being handled when the error occurred. Some errors cannot be recorded in this way and are recorded in CSR0. On receiving any interrupt, the host should always examine BABL, CERR, MISS and MERR in CSR0. If all of these bits are 0, the interrupt will be a transmit or receive interrupt (except immediately after initialisation when IDON is set).

Transmit errors

If CSR0 indicates a transmitter interrupt (TINT=1) the driver software must examine the transmit message descriptor belonging to the next buffer it expects to be transmitted. This may contain a whole packet, in which case ENP=1, or it may be the first buffer containing a packet chained over several buffers: in which case ENP=0.

The driver software must examine ERR in TMD1 to find out if the error occurred while this buffer was being transmitted. If ERR=1, the driver software can identify the nature of the error by examining the bits in TMD3. If it is the last/only buffer (ENP=1), the error should have occurred during transmission of this buffer. If it is not the last buffer (ENP=0), the error may have occurred during transmission of another buffer containing part of this packet. The driver software can find the error by examining the ERR and ENP bits of subsequent entries in the transmit message descriptor ring.

The driver software should find ERR set in one descriptor belonging to this packet. If not, the interrupt handler and LANCE are probably out of step and the contents of all buffers will have to be thrown away and the software restarted. This should never occur with correctly written software.

Receive errors

If CSR0 indicates a receiver interrupt (RINT=1) the driver software must examine the next receive message descriptor it expects to be used. This may point to a buffer containing a whole packet, in which case ENP=1, or it may point to the first buffer containing a packet chained over several buffers: in which case ENP=0.

The driver software must examine ERR in RMD1 to find out if the error occurred while this buffer was being received. If ERR=1, the driver software can identify the nature of the error by examining the other bits in RMD1. If it is the last/only buffer (ENP=1), the error should have occurred during reception of this buffer. If it is not the last buffer (ENP=0), the error may have occurred during reception of another buffer containing part of this packet. The driver software can find the error by examining the ERR and ENP bits of subsequent entries in the reception message descriptor ring.

The driver software should find ERR set in one descriptor belonging to this packet. If not, the interrupt handler and LANCE are probably out of step and the contents of all buffers will have to be thrown away and the software restarted. This should never occur with correctly written software.

6.13 Self Testing

The LANCE has comprehensive self test facilities which allow most of its features to be tested. The LANCE can be put into various test modes by setting appropriate bits in·the MODE register. The test modes include both internal and external loopback tests for testing the LANCE itself and the transmitter/receiver circuitry.

6.13.1 The MODE register

15		6							0
PROM	Undefined	INTL	DRTY	COLL	DEF	DTCR	LOOP	DTX	DRX

Figure 6.10 MODE register

PROM

The LANCE can be made to receive all packets by setting the PROM (promiscuous mode) bit. This allows the IMS B431 to monitor the amount of traffic on the network (network loading).

INTL

Valid only if LOOP = 1. If LOOP = 1 and INTL = 1, the LANCE will loopback internally. It will receive its own transmitted packets without transmitting to the Ethernet coax. If LOOP = 1 and INTL = 0, the LANCE will loopback externally. It will receive its own transmitted packets via the Ethernet coax.

DRTY (disable retry)

IF DRTY = 1, the LANCE will make only one attempt to transmit each packet, instead of the 16 attempts which it would normally make.

COLL (force collision)

Valid only in internal loopback mode: LOOP = 1, INTL = 1. COLL = 1 forces the LANCE to act as if a collision occurred on each subsequent transmission attempt. This means that 16 transmission attempts will be made for each packet (1 attempt if DRTY = 1) and a retry error will be reported in TMD3. This is used as a test of the retry and retry error circuitry.

DTCR (disable transmit CRC)

If DTCR = 0 the transmitter generates and appends a CRC to the transmitted packet. If DTCR = 1 the transmitter does not generate or append the CRC.

In loopback mode (LOOP = 1) the CRC logic can be allocated either to the transmitter (DTCR = 0) or to the receiver (DTCR = 1), but not to both. If allocated to the transmitter, the transmitter generates and appends a CRC but this is not checked or stripped by the receiver. If allocated to the receiver, the test software must append a CRC to the packet before it is transmitted as this will be checked and stripped by the receiver.

LOOP (loopback mode)

LOOP = 1 puts the LANCE into loopback mode so that transmission and reception of packets can be tested without affecting any other nodes on the network. In loopback mode, packets can be 8-32 bytes long including source address, destination address, and type field.

Internal loopback mode tests: accessing of the transmit and receive descriptor rings and buffers, and LANCE transmit and receive circuitry. External loopback tests these and also the transmit and receive paths out to the Ethernet cable and back.

Loopback tests are performed simply by placing the LANCE in loopback mode and transmitting a self-addressed packet.

DTX (disable transmitter)

If DTX = 1 the LANCE will not attempt to transmit any packets and does not access the transmit descriptor ring. DTX = 1 clears TXON.

DRX (disable receiver)

If DRX = 1 the LANCE will not receive any packets and does not access the receive descriptor ring. DRX = 1 clears RXON.

6.13.2 Loopback tests

A basic loopback test consists of

1 set up the transmit and receive descriptors and buffers

2 place the LANCE in external or internal loopback mode,

3 queue a self addressed packet for transmission,

4 allow time for the LANCE to transmit and receive the packet,

5 examine CSR0 and the transmit and receive descriptors to determine whether the test was successful,

6 identify the cause of any fault.

Internal loopback tests

Internal loopback can be used to check that the LANCE has been correctly initialised and is accessing the transmit and receive descriptor rings correctly. In internal loopback mode, the LANCE does not make any transmissions to the Ethernet and does not receive from the Ethernet.

To place the LANCE in internal loopback mode:

1 set the mode register to `#0044` (set LOOP and INTL)

2 stop the LANCE

3 initialise the LANCE.

External loopback tests

External loopback can be used to test the transmit and receive hardware including the Ethernet coax. The information provided in the TDR field of the transmit message descriptor can be used to locate faults in the Ethernet coax and drop cables.

To place the LANCE in external loopback mode:

1 set the mode register to `#0004` (set LOOP and clear INTL)

2 stop the LANCE

3 initialise the LANCE.

6.14 IMS B431 TRAM engineering data

Full details of the TRAM motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in references [1], [2], and [3]. The *Transputer Databook* [4] may also be of use. These are available as separate publications from INMOS Ltd.

6.14.1 Connectors and pin allocations

6.14.2 Pin descriptions

Pin	In/ Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
Link-speedA,B	in	Transputer link speed selection	6,7

Table 6.3 IMS B431 Pin designations

Notes:

1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.

2 Details of the physical pin locations can be found in Figure 6.11.

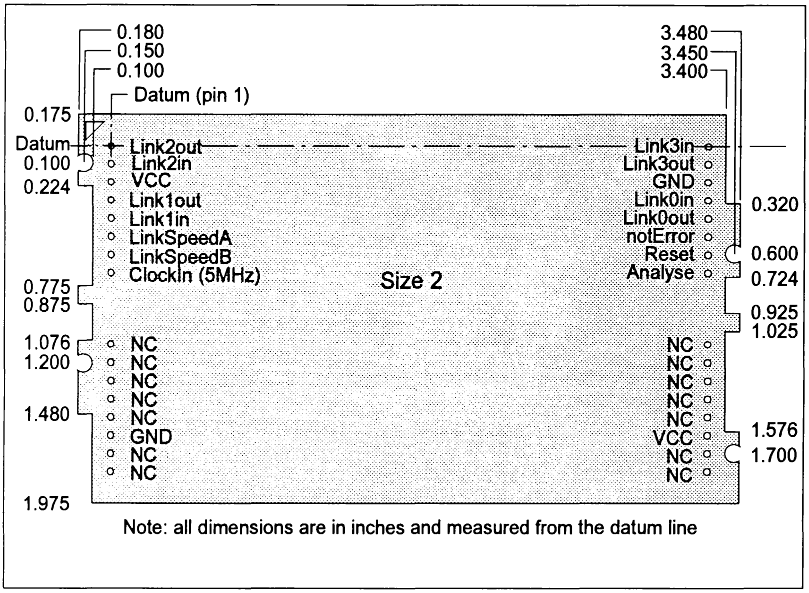

Figure 6.11 IMS B431 outline drawing (All dimensions in inches)

LinkOut0–3

Transputer link output signals. These outputs are intended to drive into transmission lines with a characteristic impedance of 100Ω. They can be connected directly to the **LinkIn** pins of other transputers or TRAMs.

LinkIn0–3

Transputer link input signals. These are the link inputs of the transputer on the IMS B431. Each input has a 10kΩ resistor to **GND** to establish the idle state, and a diode to **VCC** as protection against ESD. They can be connected directly to the **LinkOut** pins of other transputers or TRAMs.

LinkSpeedA, LinkSpeedB

These select the speeds of **Link0** and **Link1,2,3** respectively. Table 6.4 shows the possible combinations.

LinkSpeedA	LinkSpeedB	Link0	Link1,2,3
0	0	10 Mbits/s	10 Mbits/s
0	1	10 Mbits/s	20 Mbits/s
1	0	20 Mbits/s	10 Mbits/s
1	1	20 Mbits/s	20 Mbits/s

Table 6.4 Link speed selection

ClockIn

A 5MHz input clock for the transputer. The transputer synthesises its own high frequency clocks. **ClockIn** should have a stability over time and temperature of 200ppm. **ClockIn** edges should be monotonic within the range 0.8V to 2.0V with a rise/fall time of less than 8ns.

Reset

Resets the transputer, and other circuitry. **Reset** should be asserted for a minimum of 100ms. After **Reset** is deasserted a further 100ms should elapse before communication is attempted on any link. After this time, the transputer on this TRAM is ready to accept a boot packet on any of its links.

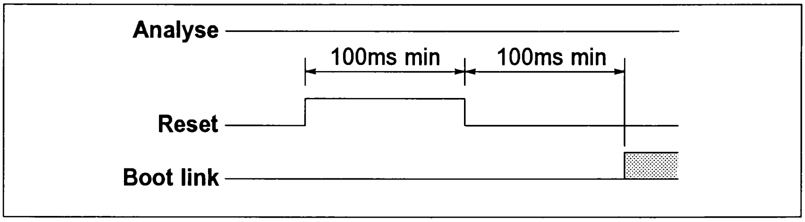

Figure 6.12 Reset timing

Analyse

This is used, in conjunction with **Reset**, to stop the transputer. It allows internal state to be examined so that the cause of an error may be determined. **Reset** and **Analyse** are used as shown in figure 6.13. A processor in analyse mode can be interrogated on any of its links.

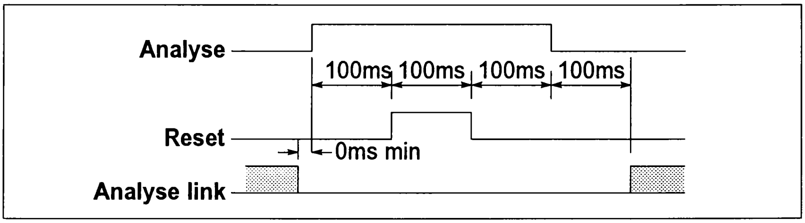

Figure 6.13 Analyse timing

notError

An open collector output which is pulled low when the transputer asserts its Error pin. **notError** should be pulled high by a 10kΩ resistor to **Vcc**. Up to 10 **notError** signals can be wired together. The combined error signal will be low when any of the contributing signals is low.

6.14.3 Memory Map

The IMS T222 on the IMS B431 has access to 64 Kbytes of memory. This is comprised of 4 Kbytes of internal transputer memory and 60 Kbytes of external SRAM. It also has memory mapped access to the two MK 7990 (LANCE) control registers. These occupy the top four byte locations in the memory map.

	Hardware byte address
IMS T222 on chip RAM	#8000 - #8FFF
External Static RAM	#9000 - #7FFB
MK7990 (LANCE) RDP	#7FFC
MK7990 (LANCE) RAP	#7FFE

Table 6.5 Memory map of the IMS B431

Table 6.5 shows the address map of the IMS B431 (the "#" sign indicates a hexadecimal number). Addresses range from #8000 through #0000 to #7FFF. The internal RAM on the IMS T222 occupies the first 4 Kbytes of address space. The MK 7990 (LANCE) occupies the top four bytes of memory. The internal RAM on the IMS T222 has a 50ns access cycle and the external SRAM has a 100ns access cycle. The LANCE has DMA to the external SRAM, DMA cycles are 600ns.

When operating at full Ethernet bandwidth the LANCE consumes approximately 38% of the available memory bus bandwidth.

6.14.4 Mechanical details

Figure 6.14 gives the vertical dimensions of an IMS B431 and Figure 2.1 is an outline drawing of the IMS B431.

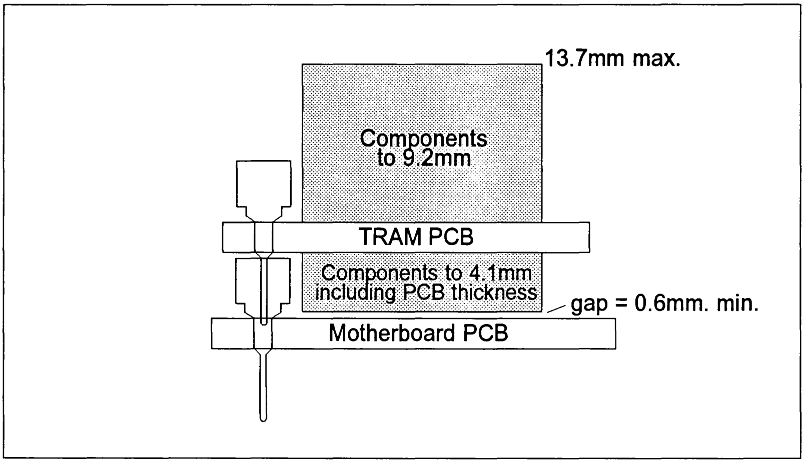

Figure 6.14 IMS B431 height specification

7 References

1 *The Transputer Development and iq Systems Databook,* INMOS Ltd, Second Edition 1991

2 *Dual-In-Line Transputer Modules (TRAMs)* , INMOS Technical Note 29, also in [1].

3 *Module Motherboard Architecture,* INMOS Technical Note 49, also in [1].

4 *Transputer Databook,* INMOS Ltd, Second Edition 1989.

5 *Connecting INMOS links*, INMOS Technical Note 18, INMOS Ltd 1987.

6 *ANSI/IEEE 802.3–1984 Local Area Networks, Carrier Sense Multiple Access with Collision Detection (CSMA/CD). Access Method and Physical Layer Specifications,* ANSI/IEEE 802.3–1984.

7 *ANSI/IEEE 802.3a,b,c,e, supplement to ANSI/IEEE 802.3–1984,* ANSI/IEEE 1988.

Appendices

A Directory structure

IMS F006A files are installed within the following directory structure:–

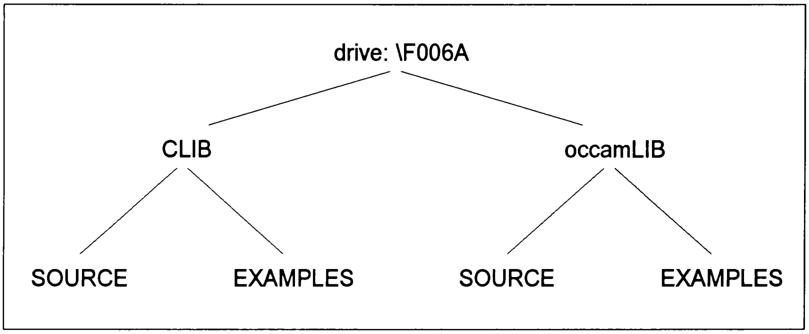

Figure A.1 IMS F006A directory structure

Which, after a successful installation, should contain the following files:–

drive:**\F006A**

```
b431drvr.lku
```

drive:**\F006A\CLIB**

```
b431.lib
b431io.h
b431load.h
b431test.h
```

drive:**\F006A\CLIB\SOURCE**

```
b431drvr.h
b431io.c
b431load.c
b431test.c
imsb431.h
```

drive:**\F006A\CLIB\EXAMPLES**

```
listen.c
listen.cfs
listen.lnk
loopback.c
loopback.cfs
loopback.lnk
```

drive: \F006A\OCCAMLIB

```
b431.lib
b431.liu
b431io.inc
b431load.inc
b431test.inc
```

drive: \F006A\OCCAMLIB\SOURCE

```
b431drvr.inc
b431io.occ
b431load.occ
b431test.occ
imsb431.inc
marshall.occ
```

drive: \F006A\OCCAMLIB\EXAMPLES

```
listen.occ
listen.pgm
loopback.occ
loopback.pgm
txbench.occ
txbench.pgm
```

B Example programs

B.1 IMS F006A example program

The following examples can be compiled and run on the hardware configuration described in section 3.5.1. Source code for the ANSI C example can be found in the directory `\F006A\CLIB\EXAMPLES` and for the equivalent occam example, in `\F006A\OCCAMLIB\EXAMPLES`.

The example is a simple Ethernet listener. It initialises the Ethernet interface for promiscuous mode packet reception and then displays the header segment of every packet received. Periodically it also requests and prints out Ethernet statistics accumulated by the device driver. The example demonstrates the use of the IMS F006A interface library and how to configure a transputer network incorporating a single IMS B431 Ethernet TRAM.

B.1.1 C Example

See the files:- `listen.c`, `listen.cfs` and `listen.lnk` in the ANSI C examples directory. To compile and then run the example, type:–

1 `imakef listen.btl /c`. (Ignore the message warning about `b431drvr.lku` having an unknown file extension type. The device driver has the `.lku` extension to stop `imakef` generating rules to create it).

2 `make /f listen.mak`

3 `iserver /se /sb listen.btl`

listen.c:

```
/* IMS F006A example program
   Copyright INMOS Limited 1991 */

/* Simple ethernet listener. This program initialises the IMS B431
   ethernet interface in promiscuous receive mode and prints out the
   header segment of every packet it receives. It also periodically
   requests and displays accumulated ethernet statistics as gathered
   by the IMS B431 device driver. */

#include <time.h>
#include <misc.h>
#include <stdio.h>
#include <stdlib.h>
#include <b431io.h>
#include <process.h>

/* A function to continuously request ethernet statistics,
   once every 20 seconds, this uses the channel to the IMS B431
   device driver and will be executed as a transputer process */

void request_stats(
      Process      *p,
      Channel *to_b431 )
{
        p = p; /* Gets rid of compiler warning message */

        while ( 1 )

        {
                ProcWait( (int) CLOCKS_PER_SEC * 20 );
                B431_Ether_Stats( to_b431 );
        }
}


/* Print the header segment of a packet */

void print_packet_header(
        unsigned char *ethernet_packet,
        int             ethernet_packet_length)
{
        int     i;

        printf( "Packet received, length = %d bytes\n[ ",
                ethernet_packet_length);

        for ( i = 0; i < PACKET_HEADER_SIZE; i++ )
                printf( "0x%02x ", ethernet_packet[i] );
        printf( "]\n" );
}

/* print the ethernet statistics structure */

void print_ether_stats( ETHER_STATS ether_stats )
{
        printf( "ethernet statistics:\n" );
        printf( "tx_packets = %lu, ", ether_stats.tx_packets );
        printf( "rx_packets = %lu\n", ether_stats.rx_packets );
        printf( "framing_errors = %lu, ", ether_stats.framing_errors );
        printf( "crc_errors = %lu\n", ether_stats.crc_errors );
        printf( "packets_dropped = %lu, ", ether_stats.packets_dropped );
        printf( "packets_missed = %lu\n", ether_stats.packets_missed );
        printf( "deferred_transmits = %lu, ", ether_stats.deferred_transmits) ;
        printf( "late_collisions = %lu, ", ether_stats.late_collisions );
        printf( "carrier_lost = %lu\n", ether_stats.carrier_lost );
        printf( "failed_retries = %lu, ", ether_stats.no_more_retries );
        printf( "single_retries = %lu, ", ether_stats.single_retries );
        printf( "many_retries = %lu\n", ether_stats.multiple_retries );
        printf( "average_tdr_value = %lu\n", ether_stats.average_tdr_value );
}

/* Initialise the ethernet interface and then start it running.
   Continuously call B431_Waitfor_Event() to receive ethernet packets
   and print their header. Start a parallel process to request ethernet
   statistics and print them */

int main()
{
        Process         *p;
        char    result;
        ETHER_STATS     ether_stats;
        /* Arbitrary physical address */
        unsigned char physical_address[PHYSICAL_ADDRESS_SIZE]=
                        { 0x22, 0x44, 0x88, 0x88, 0x44, 0x22 };
        /* Logical address filter disables multicast packet reception */
        unsigned char logical_address_filter[LOGICAL_ADDRESS_FILTER_SIZE]=
                        { 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 };
        int     error_code,receive_packet_length;
        unsigned char receive_packet[MAX_PACKET_LENGTH];
        unsigned char failed_packet_data[FAILED_PACKET_LENGTH];
        Channel         *to_b431, *from_b431, *cancel = ChanAlloc();

        /* Channels to / from IMS B431 device driver are obtained from
```

```
           the configuration environment */

        to_b431 = (Channel *) get_param( 4 );
        from_b431 = (Channel *) get_param( 3 );
        printf( "IMS-F006 ethernet listener\n" );

        /* Initialise and start the ethernet interface in promiscuous
           packet receive mode */

        result = B431_Init_Normal( from_b431, to_b431,
                        physical_address, logical_address_filter,
                        MEMORY_CHECK | PROMISCUOUS_RX );
        if ( result != INIT_SUCCESS )
        {
                printf( "B431_Init_Normal() failed = %d\n", result );
                exit_terminate(0);
        }
        result = B431_Start_Ether( from_b431, to_b431 );
        if ( result != START_SUCCESS )
        {
                printf( "B431_Start_Ether() failed = %d\n", result );
                exit_terminate(0);
        }

        /* Allocate and start a parallel process to request ethernet
           statistics once every 20 seconds. It is passed a single parameter,
           the channel to the B431 device driver */

        p = ProcAlloc( ( void (*)() ) request_stats, 0, 1, to_b431 ) ;
        ProcRun( p );

        /* Wait for ethernet interface activity. Print ethernet statistics
           or the headers of packets received. Ignore everything else */

        while ( 1 )
        {
                result = B431_Waitfor_Event( from_b431, cancel,
                                &ether_stats,
                                receive_packet, &receive_packet_length,
                                &error_code, failed_packet_data );

                switch (result)
                {
                        case B431_ETHER_STATS:
                                print_ether_stats( ether_stats );
                                break;

                        case B431_RX_PACKET:
                        print_packet_header( receive_packet,
                                             receive_packet_length );

                                break;
                }
        }
}
```

listen.cfs:

```
/* IMS F006A example program
   Copyright INMOS Limited 1991 */

/* Configuration file for simple ethernet listener. */

/* Hardware description. Configuration consists of two TRAMs arranged
   in a 2-1 pipeline. The first is a 2 Mbyte IMS B404 TRAM connected to
   the host. It is also connected to the second, IMS B431, ethernet TRAM. */

T800 (memory = 2M) root;
/* IMS B404 2Mbyte TRAM */
T212 (memory = 64K) b431;
/* IMS B431 ethernet TRAM */

connect root.link[1] to host;
connect root.link[2] to b431.link[1];

/* Software description. Configure the ethernet listener example
   program to run on the (root) IMS B404 TRAM. The ethernet device driver
   is placed on the IMS B431 ethernet TRAM. */

/* Root processor runs example listen program. Connect it to the host
   server and also the IMS B431 device driver */

input from_server;   /* input edge */
output to_server;    /* output edge */

process ( stacksize = 8K, heapsize = 50K,
        interface   (input fs, output ts,
                     input from_b431, output to_b431 ) ) listen;

connect from_server to listen.fs;
connect to_server to listen.ts;
/* Connect to server */

/* IMS B431 ethernet TRAM runs the device driver. Connect it to the
   other IMS B404 TRAM which runs the listen program */

process ( interface ( input to_b431,
                     output from_b431 ) ) b431_driver;

connect listen.to_b431 to b431_driver.to_b431;
connect listen.from_b431 to b431_driver.from_b431;

/* Select linked units for each process and map to their respective
   processor. Map channels to their respective links */

use "listen.c8x" for listen;/* Example listen program */
use "b431drvr.lku" for b431_driver;/* IMS B431 device driver */

place listen on root;
place b431_driver on b431;
/* Place processes */

place from_server on host;
place to_server on host;
/* Place host channels */
```

B.1.2 occam Example

See the files:- `listen.occ` and `listen.pgm` in the occam examples directory. To compile and then run the example, type:–

1. `imakef listen.btl /y`. (Ignore the message warning about `b431drvr.lku` having an unknown file extension type. The device driver has the `.lku` extension to stop `imakef` generating rules to create it).

2. `make /f listen.mak`

3. `iserver /se /sb listen.btl`

listen.occ:

```
--
-- IMS F006A example program
-- Copyright INMOS Limited 1991
--
-- Simple ethernet listener. This program initialises the IMS B431
-- ethernet interface in promiscuous receive mode and prints out the
-- header segment of every packet it receives. It also periodically
-- requests and displays accumulated ethernet statistics as gathered
-- by the IMS B431 device driver.
--

-- Toolset header and library files.

#INCLUDE "ticks.inc"
#INCLUDE "hostio.inc"
#USE "hostio.lib"

-- Main entry point has an ISERVER channel pair for connection to the
-- host and a further channel pair for communication with the IMS B431
-- TRAM device driver.

PROC listen( CHAN OF SP fs, ts,
             CHAN OF ANY from.b431, to.b431 )

  -- IMS F006A header and library files

  #INCLUDE "b431io.inc"
  #USE "b431.lib"

  -- Variables

  -- Arbitrary physical address
  VAL [PHYSICAL.ADDRESS.SIZE]BYTE physical.address IS
    "*#22*#44*#88*#88*#44*#22" :

  -- Logical address filter disables multicast packet reception
  VAL [LOGICAL.ADDRESS.FILTER.SIZE]BYTE logical.address.filter IS
    "*#00*#00*#00*#00*#00*#00*#00*#00" :

  -- For use with B431.Waitfor.Event() etc.
  BYTE result :
  CHAN OF BYTE abort :
  [ETHER.STATS.SIZE]INT32 ethernet.stats :
  INT error.code, receive.packet.length :
  [MAX.PACKET.LENGTH]BYTE receive.packet :
  [FAILED.PACKET.LENGTH]BYTE failed.packet.data :
```

```
-- Initialise the ethernet interface and then start it running.
-- Continuously call B431.Waitfor.Event() to receive ethernet packets
-- and print out the header. Request and print out ethernet statistics
-- every 20 seconds.
SEQ

  so.write.string.nl( fs, ts, "IMS-F006 ethernet listener" )

  -- Initialise ethernet interface in promiscuous receive mode and
  -- perform packet buffer memory check.
  B431.Init.Normal( from.b431, to.b431,
                    physical.address, logical.address.filter,
                    MEMORY.CHECK \/ PROMISCUOUS.RX, result )
  IF
    ( result <> INIT.SUCCESS )
      so.write.string.nl( fs, ts, "B431.Init.Normal() failed" )
    TRUE
      SKIP

  -- Start the ethernet interface running.
  B431.Start.Ether( from.b431, to.b431, result )
  IF
    ( result <> START.SUCCESS )
      so.write.string.nl( fs, ts, "B431.Start.Ether() failed" )
    TRUE
      SKIP

  -- Fork into two separate parallel processes. One will receive
  -- and print out either ethernet packet headers or ethernet statistics
  -- when received from the device driver. The other will request ethernet
  -- statistics once every 20 seconds.

  PAR

    -- Process to request ethernet statistics once every 20 seconds.

    WHILE TRUE
      TIMER clock :
      INT time.now :
      SEQ
        clock ? time.now
        clock ? AFTER time.now PLUS ( (INT lo.ticks.per.second) * 20 )
        B431.Ether.Stats( to.b431 )

    -- Process to print out ethernet packet headers or statistics when
    -- received from the IMS B431 device driver.

    WHILE TRUE
      SEQ

        -- Wait for the ethernet interface to do something.
        B431.Waitfor.Event( from.b431, abort, ethernet.stats,
                            receive.packet, receive.packet.length,
                            error.code, failed.packet.data, result )

        CASE result

          -- The other process requested ethernet statistics, print
          -- out each field.
          B431.ETHER.STATS
            VAL []INT ethernet.stats RETYPES ethernet.stats :
            SEQ
              so.write.string.nl( fs, ts, "ethernet statistics:" )
              so.write.string( fs, ts, "tx.packets = " )
              so.write.int( fs, ts, ethernet.stats[STATS.TX.PACKETS], 0 )
              so.write.string( fs, ts, ", rx.packets = " )
          so.write.int( fs, ts, ethernet.stats[STATS.RX.PACKETS], 0 )
          so.write.nl( fs, ts )
          so.write.string( fs, ts, "framing.errors = " )
          so.write.int( fs, ts, ethernet.stats[STATS.FRAMING.ERRORS], 0 )

          so.write.string( fs, ts, ", crc.errors = " )
          so.write.int( fs, ts, ethernet.stats[STATS.CRC.ERRORS], 0 )
          so.write.nl( fs, ts )
          so.write.string( fs, ts, "packets.dropped = " )
          so.write.int( fs, ts, ethernet.stats[STATS.PACKETS.DROPPED], 0)

          so.write.string( fs, ts, ", packets.missed = " )
          so.write.int( fs, ts, ethernet.stats[STATS.PACKETS.MISSED],
          so.write.nl( fs, ts )

          so.write.string( fs, ts, "deferred.transmits = " ) 0 )
          so.write.int( fs, ts, ethernet.stats[STATS.DEFERRED.TRANSMITS], 0 )
          so.write.string( fs, ts, ", late.collisions = " )
          so.write.int( fs, ts, ethernet.stats[STATS.LATE.COLLISIONS], 0)
          so.write.string( fs, ts, ", carrier.lost = " )
          so.write.int( fs, ts, ethernet.stats[STATS.CARRIER.LOST], 0 )
          so.write.nl( fs, ts )
          so.write.string( fs, ts, "no.more.retries = " )
          so.write.int( fs, ts, ethernet.stats[STATS.NO.MORE.RETRIES], 0)
          so.write.string( fs, ts, ", single.retries = " )
          so.write.int( fs, ts, ethernet.stats[STATS.SINGLE.RETRIES], 0 )
          so.write.string( fs, ts, ", multiple.retries = " )
          so.write.int( fs, ts, ethernet.stats[STATS.MULTIPLE.RETRIES], 0)
          so.write.nl( fs, ts )
          so.write.string( fs, ts, "average.tdr.value = " )
          so.write.int( fs, ts, ethernet.stats[STATS.AVERAGE.TDR.VALUE], 0)
          so.write.nl( fs, ts )

      -- An ethernet packet arrived, print out it's header segment.
      B431.RX.PACKET
        SEQ
          so.write.string( fs, ts, "Packet received, length = " )
          so.write.int( fs, ts, receive.packet.length, 0 )
          so.write.string.nl( fs, ts, " bytes" )
          so.write.string( fs, ts, "[ " )
          SEQ i = 0 FOR PACKET.HEADER.SIZE
            SEQ
              so.write.hex.int( fs, ts, INT receive.packet[i], 2 )
              so.write.string( fs, ts, " " )
          so.write.string.nl( fs, ts, "]" )

      -- Ignore everything else.
      ELSE
        SKIP

:
```

listen.pgm:

```
--
-- IMS F006A example program
-- Copyright INMOS Limited 1991
--
-- Configuration file for simple ethernet listener.
--

-- Hardware description. Configuration consists of two TRAMs arranged
-- in a 2-1 pipeline. The first is a 2 Mbyte IMS B404 TRAM connected to
-- the host. It is also connected to the second, IMS B431, ethernet TRAM.

VAL K IS 1024 :
VAL M IS K * K :

ARC HostLink :
NODE root, b431 :
NETWORK ethernet.listener
  DO
    SET root (type, memsize := "T800", 2 * M)  -- IMS B404 2Mbye TRAM
    SET b431 (type, memsize := "T212", 64 * K)  -- IMS B431 ethernet TRAM
    CONNECT root[link][2] TO b431[link][1]
    CONNECT root[link][1] TO HOST WITH HostLink
:

-- Software description. Configure the ethernet listener example
-- program to run on the (root) IMS B404 TRAM. The ethernet device driver
-- is placed on the IMS B431 ethernet TRAM.

#INCLUDE "hostio.inc"

#USE "listen.c8h"  -- ethernet listener example program
#USE "b431drvr.lku"  -- IMS B431 ethernet TRAM device driver

CONFIG

  CHAN OF SP root.to.host, host.to.root :
  CHAN OF ANY root.to.b431, b431.to.root :
  PLACE root.to.host, host.to.root ON HostLink :

  PLACED PAR

    -- Example program placed on the IMS B404
    PROCESSOR root
      listen( host.to.root, root.to.host, b431.to.root, root.to.b431 )

    -- ethernet device driver placed on the IMS B431
    PROCESSOR b431
      INMOS.B431.Driver( root.to.b431, b431.to.root )
:
```

Sales Offices

EUROPE

DENMARK

2730 HERLEV
Herlev Torv, 4
Tel. (45–42) 94.85.33
Telex:35411
Telefax: (45–42) 948694

FINLAND

LOHJA SF–08150
Karjalankatu, 2
Tel. 12.155.11
Telefax: 12.155.66

FRANCE

94253 GENTILLY Cedex
7, Avenue Gallieni – BP 93
Tel. (33–1) 47.40.75.75
Telex: 632570 STMHQ
Telefax: (33–1) 47.40.79.10

67000 STRASBOURG
20, Place des Halles
Tel. (33) 88.75.50.66
Telex: 870001F
Telefax: (33) 88.22.29.32

GERMANY

6000 FRANKFURT
Gutleutstrasse, 322
Tel. (49–69) 237492
Telex: 176997 689
Telefax: (49–69) 231957
Teletex:6997689=STVBP

8011 GRASBRUNN
Bretonischer Ring, 4
Neukerloh Technopark
Tel. (49–89) 46006–0
Telex: 528211
Telefax: (49–89) 4605454
Teletex:897107=STDISTR

5000 HANNOVER 51
Rotenburgerstrasse, 28A
Tel. (49–511) 615960
Telex: 175118418
Telefax: (49–511) 6151243

8500 NURNBERG 20
Erlenstegenstrasse, 72
Tel. (49–911) 59893–0
Telex: 626243
Telefax: (49–911) 5980701

5200 SIEGBURG
Frankfurter Str. 22a
Tel. (49–2241) 660 84–86
Telex: 889510
Telefax: (49–2241) 67584

7000 STUTTGART
Oberer Kirchhaldenweg, 135
Tel. (49–711) 692041
Telex: 721718
Telefax: (49–711) 691408

ITALY

20090 ASSAGO (MI)
V.le Milanofiori – Strada 4 – Palazzo A/4/A
Tel. (39–2) 89213.1 (10 lines)
Telex: 330131 – 330141 SGSAGR
Telefax: (39–2) 8250449

40033 CASALECCHIO DI RENO (BO)
Via R. Fucini, 12
Tel. (39–51) 591914
Telex: 512442
Telefax: (39–51) 591305

00161 ROMA
Via A. Torlonia, 15
Tel. (39–6) 8443341
Telex: 620653 SGSATE I
Telefax: (39–6) 8444474

NETHERLANDS

5652 AM EINDHOVEN
Meerenakkerweg, 1
Tel. (31–40) 550015
Telex: 51186
Telefax: (31–40) 528835

SPAIN

08021 BARCELONA
Calle Platon, 6, 4th Floor, 5th Door
Tel. (34–3) 4143300 – 4143361
Telefax: (34–3) 2021461

28027 MADRID
Calle Albacete, 5
Tel. (34–1) 4051615
Telex: 27060 TCCEE
Telefax: (34–1) 4031 134

SWEDEN

S–16421 KISTA
Borgarfjordsgatan, 13 – Box 1094
Tel. (46–8) 7939220
Telex: 12078 THSWS
Telefax: (46–8) 7504950

SWITZERLAND

1218 GRAND–SACONNEX (GENEVA)
CheminFrançois-Lehmann18/A
Tel. (41–22) 7986462
Telex: 415493 STM CH
Telefax: (41–22) 7984869

United Kingdom And Eire

MARLOW, BUCKS SL7 1YL
Planar House, Parkway
Globe Park
Tel. (44–628) 890800
Telex: 847458
Telefax: (44–628) 890391

AMERICAS

BRAZIL

05413 SAO PAULO
R. Henrique Schaumann 286–CJ33
Tel. (55–11) 883–5455
Telex: (391) 11–37988 "UMBR BR"
Telefax: 11–551–128–22367

CANADA

BRAMPTON, ONTARIO
341, Main St. North
Tel. (416) 455–0505
Telefax:416–455–2606

USA

NORTH & SOUTH AMERICAN MARKETING HEADQUARTERS
1000, East Bell Road
Phoenix, AZ 85022
(1)–(602)867–6100

SALES COVERAGE BY STATE

ALABAMA
303, Williams Avenue,
Suite 1031,
Huntsville, AL 35801–5104
Tel.(205)533–5995

ARIZONA
1000, East Bell Road
Phoenix, AZ 85022
Tel. (602) 867–6100

CALIFORNIA
200 East Sandpointe,
Suite 120,
Santa Ana, CA 92707
Tel. (714) 957–6018

2055, Gateway Place,
Suite 300
San José, CA 95110
Tel. (408) 452–9122

COLORADO
1898, S. Flatiron Ct.
Boulder, CO 80301
Tel.(303)449–9000

FLORIDA
902 Clint Moore Road
Congress Corporate Plaza II
Bldg. 3 – Suite 220
Boca Raton, FL 33487
Tel.(407)997–7233

GEORGIA
6025, G.Atlantic Blvd.
Norcross, GA 30071
Tel. (404) 242 –7444

ILLINOIS
600, North Meacham
Suite 304,
Schaumburg, ILL 60173–4941
Tel. (708) 517–1890

INDIANA
1716, South Plate St.
Kokomo, IN 46902
Tel. (317) 459–4700

MASSACHUSETTS
55, Old Bedford Road
Lincoln North
Lincoln, MA 01773
Tel. (617) 259–0300

MICHIGAN
17197, N. Laurel Park Drive
Suite 253,
Livonia, MI 48152
Tel. (313) 462–4030

MINNESOTA
7805, Telegraph Road
Suite 112
Bloomington, MN 55438
Tel. (612) 944–0098

NEW JERSEY
Staffordshire Professional Ctr.
1307, White Horse Road Bldg. F.
Voorhees, NJ 08043
Tel. (609) 772–6222

NEW YORK
2–4, Austin Court
Poughkeepsie, NY 12603–3633
Tel. (914) 454–8813

NORTH CAROLINA
4505, Fair Meadow Lane
Suite 220
Raleigh, NC 27607
Tel. (919) 787–6555

TEXAS
1310, Electronics Drive
Carrollton, TX 75006
Tel. (214) 466–7402

ASIA/PACIFIC

AUSTRALIA

NSW 2027 EDGECLIFF
Suite 211, Edgecliff Centre
203–233, New South Head Road
Tel. (61–2) 327.39.22
Telex: 071 126911 TCAUS
Telefax: (61–2) 327.61.76

HONG KONG

WANCHAI
22nd Floor – Hopewell Centre
183, Queen's Road East
Tel. (852–5) 8615788
Telex: 60955 ESGIES HX
Telefax: (852–5) 8656589

INDIA

NEW DELHI 110001
Liaison Office
62, Upper Ground Floor
World Trade Centre
Barakhamba Lane
Tel. 3715191
Telex: 031–66816 STMI IN
Telefax: 3715192

KOREA

SEOUL 121
8th Floor Shinwon Building
823–14, Yuksam-Dong
Kang-Nam-Gu
Tel. (82–2) 553–0399
Telex: SGSKOR K29998
Telefax: (82–2) 552–1051

MALAYSIA

PULAU PINANG 10400
4th Floor, Suite 4–03
Bangunan FOP, 123D Jalan Anson
Tel. (04) 379735
Telefax: (04) 379816

SINGAPORE

SINGAPORE 2056
28 Ang Mo Kio – Industrial Park, 2
Tel. (65) 48214 11
Telex: RS 55201 ESGIES
Telefax: (65) 4820240

TAIWAN

TAIPEI
12th Floor
571, Tun Hua South Roa
Tel. (886–2) 755–41 11
Telex:10310 ESGIE TW
Telefax: (886–2) 755–4008)

JAPAN

TOKYO 108
Nisseki Takanawa Bld. 4F
2–18–10 Takanawa
Minato-ku
Tel. (81–3) 3280–4125
Telefax: (81–3) 3280–4131