SOFTWARE COMPONENT SPECIFICATION

SYSTEM:	Series 6
SUBSYSTEM:	Local Area Network Controller
COMPONENT:	MAC Firmware
PLANNED RELEASE:	MOD400 Release 4.0
SPECIFICATION REVISION NUMBER:	1
DATE:	July 26th 1985
AUTHOR:	Richard M. Collins

This specification describes the current definitions of the subject software component, and may be revised in order to incorporate design improvements.

TABLE OF CONTENTS

REFERENCES

1.	09-0016-00	ESPL Software Technical Reference Manual, Vol. 1, Kernel and Support Software (Bridge Communications, Inc.)
2.	60149817	LAN Software EPS-1
3.	60149766	Local Area Controller Subsystem (LACS) EPS-1
4.	60149824	DPS6 Local Area Network Controller PFS
5.	--------	LACS Logic Block Diagrams, Release 2.0
6.	DSA-41	Local Area Networks
7.	IEEE 802.1	Local Area Network Overview
8.	IEEE std 802.2	Logical Link Control, 1985
9.	IEEE std 802.3	CSMA/CD, 1985
10.	IEEE std 802.4	Token Bus, 1985
11.	IEEE std 802.5	Token Ring, 1985
12.	ISO 7498	Open System Interconnection Reference Model
13.	--------	Ethernet Specification, Version 2.0 (Digital Equipment Corp., Intel Corp., Xerex Corp.)
14.	MC68000UM (AD2)	MC68000 User´s Manual (Motorola Inc.)
15.	03378D	AM7990 Reference Manual (Advanced Micro Devices Inc.)
16.	--------	AM7990 Errata Document, 1984 (AMD Inc.)

ABBREVIATIONS/DEFINITIONS

MAC	-	Media Access Control
DMA	-	Direct Memory Access. No intervention from the local processor. Data Transferes takes place between peripheral and memory
CSMA/CD	-	Carrier Sense Multiple Access/Collision Detection
ECB	-	Ethernet Control Block
LAN	-	Local Area Network
LACS	-	Local Area Controller Subsystem
LLC	-	Logical Link Control
FW	-	Firmware
Kernel	-	Core group of service routines making up a state-of-the-art high performance real time operating system
RX	-	MAC specific receive process
TX	-	MAC specific transmit process
LM	-	MAC specific local layer manager process
PRIVATE	-	Catagory of memory known to the Kernel. It consists of the bank of memory local to the 68000 processor on the LACS
SHARED	-	Catagory of memory known to the Kernel. It consists of the bank of memory local to the MAC adapters on the LACS

SCOPE

This document defines the entire functionality of the MAC firmware module

1 INTRODUCTION AND OVERVIEW

1.1 Background

This module was designed and coded by the hardware development group responsible for the LACS. It represents second generation firmware. It was written in the "C" programming language as a specialized extension of the kernel. The module implements traditional firmware by performing hardware specific duties, but it also implements the MAC service interface. This interface is characterized and defined by the 802 family of standards.

1.2 Basic Purpose

The purpose of this module is to provide MAC-user software with the service primitives defined in the 802 family of LAN standards. The primitives allow a user to request transmission, get a confirmation that the transmission took place, and to get an indication when data has been received from the LAN. In providing these services the module must hide the details of the various MACs that can be attached to the LACS as well as from the details and intricacies of the chips that implement a given MAC.

1.3 Basic Structure

The basic structure of the MAC firmware subsystem and of the ethernet specific firmware is diagramed on the following two pages.

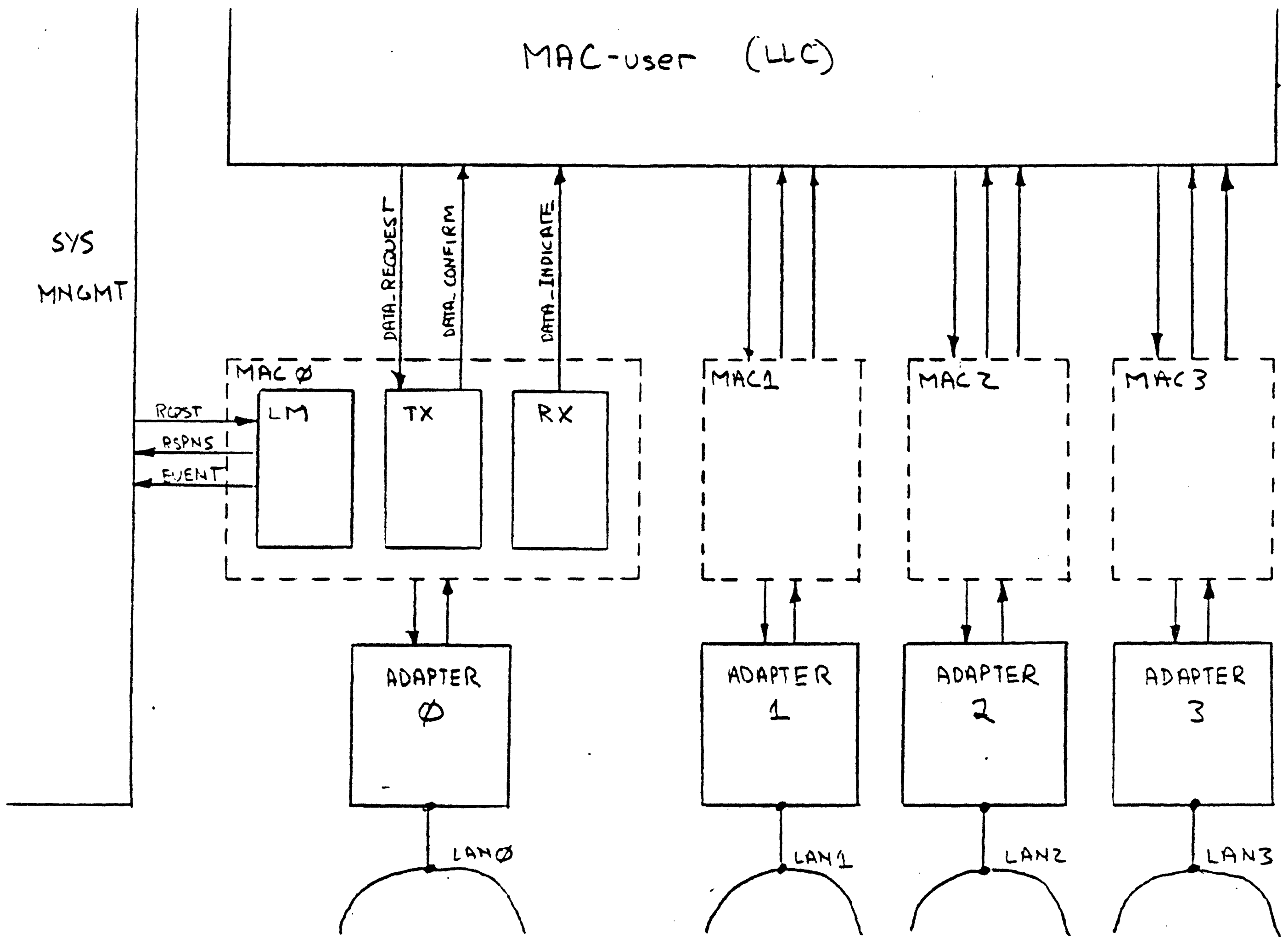

FIGURE 1
OVERALL MAC FIRMWARE SUBSYSTEM

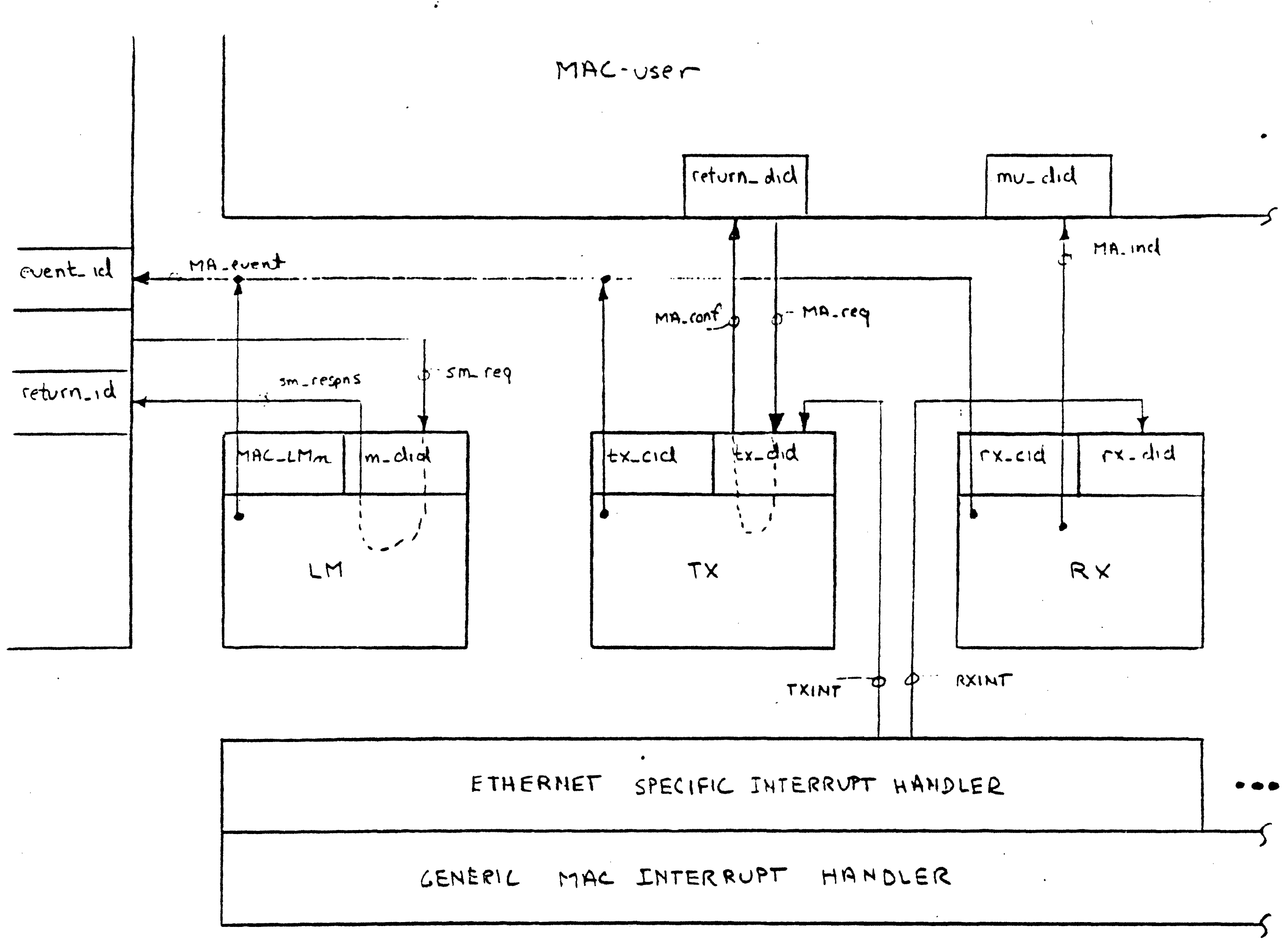

FIGURE 2
ETHERNET SPECIFIC MAC

1.4 Basic Operation

An individual MAC is basically made up of a transmit, receive, and layer management module. These modules are normally in a suspended state waiting on the arrival of a message into their data mailbox. For transmission, the MAC-user will send an MA.request message to the transmit module. This will cause the transmit module to become active. It will append the MAC header to the frame and queue it for transmission by the protocol chip. The chip will generate an interrupt upon completion of the transmission. The MAC specific interrupt handler will then send a transmit interrupt (TXINT) message back to the transmit module. The transmit module will then compose an MA.confirmation message and send it back to the MAC-user´s return mailbox. For receive, the MAC specific interrupt handler will send a receive interrupt (RXINT) message to the receive module. The receive module will allocate new buffers to replace those used by the incoming packet. It will then strip the MAC header from the frame and compose an MA.indication message which it then sends to the MAC-user´s data mailbox.

2 EXTERNAL SPECIFICATIONS

2.1 Owned Data Structures

2.1.1 Generic MAC

The generic MAC owns a data structure call MAC_HEAD. This structure contains pointers to the MAC specific data areas and the MAC specific interrupt handlers for each of the four possible MAC adapters that can be attached to the LACS.

```
/*
 *      MAC_HEAD global STRUCTURE
 */
#define MAC_HEAD struct mac_head
struct mac_head {
        int     (*int_proto[4])();      /* func ptrs to int proc */
        int     *proto_data[4];         /* ptrs to the data area */
};

MAC_HEAD        mac;    /* mac data struc for all access methods */
```

2.1.2 Ethernet Specific MAC

Each ethernet specific MAC process (one per adapter) owns an Ethernet Control Block (ecb). It is accessed by the RX, TX and LM modules. It is allocated from PRIVATE storage at run Time. This allows multiple copies of the ethernet group of modules to be running at the same time. Each instance will allocate a unique block of memory for its ecb.

```
/*
 *      ECB (Ethernet Control Block) STRUCTURE
 */

#define ECB struct ecb
struct ecb {
        short   state;          /* state of this mac layer */

        LAN_ADR ether_SA;       /* source address this node */

        caddr_t am79_rap;       /* register address port */
        caddr_t am79_rdp;       /* register data port */

        struct init79   *init_base;     /* ptr to Lance init blk */
```

```
          /* Transmit related data area */
          MSG     *m_head;          /* fwd ptr to pending tx msg que */
          MSG     *m_tail;          /* ptr to last tx msg in que */

          CFTxD   *txring_strt;     /* base address tx ring */
          CFTxD   *txring_max;      /* last address tx ring */
          CFTxD   *txring_now;      /* current position in tx ring */
          HFTxD   rtrv_txdesc[3];   /* work space to retrv tx descs */

          char    txr_leng;         /* value of tx ring length */
          char    tlen;             /* bit field size of tx length */

          /* reideve related data area */
          BD      *bd_fst;          /* ptr to top bd on rx ring */
          BD      *bd_lst;          /* ptr to bottom bd on ring */

          CFRxD   *rxring_strt;     /* base address rx ring */
          CFRxD   *rxring_max;      /* last address rx ring */
          CFRxD   *rxring_now;      /* current position in rx ring */
          HFRxD   rtrv_rxdesc[3];   /* work space to retrv rx descs */
          HFRxD   nxt_rxdesc[3];    /* space for pre-allocated descs */

          char    rxr_leng;         /* value of rx ring length */
          char    rlen;             /* bit field size of rx length */

          /* mailbox ids for communication with upper layers */
          MBID    tx_cid;           /* control mboxid of mac tx proc */
          MBID    tx_did;           /* data mboxid of mac tx proc */

          MBID    rx_cid;           /* control mboxid of mac rx proc */
          MBID    rx_did;           /* data mboxid of mac rx proc */

          MBID    lm_cid;           /* control mboxid of layer mngmt */
          MBID    lm_did;           /* data mboxid of layer mngmnt */

          MBID    mu_did;           /* MAC user's (LLC) data mbox id */
          MBID    event_did;        /* MBID to send event .ind's */

     /* The following two SAPs are not available with 802.3
      *   MBID    ir_did                Immediate responce SAP,(802.4)
      *   MBID    sm_did                System Managment data SAP
      */

          /* miscellaneous data items */
          AMSG    *alarm_msg;       /* pointer to message for alarm */
          short   activity_flag;    /* tx in progress flag */
          short   bufsiz;           /* size of buffers in use */
          struct  ether_stats stats;        /* status structure */
     };
```

```
/*
 *      ether_stats STRUCTURE
 */

struct ether_stats {
        short   framerr;        /* # of framing errors on rx */
        short   crcerr;         /* # of crc errors on receive */
        short   badpkt;         /* # of packets lost by 7990 */
        short   lostpkt;        /* # of packets lost by mac fw */
        short   rx_fatal;       /* # of rx initiated chip resets */
        short   tx_fatal;       /* # of tx initiated chip resets */
        short   coll;           /* # of collisions */
        short   reset;          /* # of chip resets */
        short   retry;          /* # of retrys */
        short   packets;        /* # of packets sent */
        short   lm_calls;       /* # of times lm called */
        short   bad_sendmsg;    /* # of failed sendmsg calls */
};


/*
 *      am7990 initialization block definition
 */
typedef struct init79 {
        ushort  mode;           /* hardware mode word */
        LAN_ADR node_SA;        /* source address this node */
        ushort  ladrf1;         /* logical address filter */
        ushort  ladrf2;
        ushort  ladrf3;
        ushort  ladrf4;
        ADDRESS rx_ring;        /* rcv desc ring point + length */
        ADDRESS tx_ring;        /* tx desc ring pointer + length */
};
```

2.1.3 Token Bus Specific MAC

TBD

2.1.4 Token Ring Specific MAC

TBD

2.1.5 Starlan Specific MAC

TBD

2.2 External Interfaces

External communication with this module is via Kernel mailbox messages. There are three types of messages that are used for normal communication, the MA_req, the MA_conf, and the MA_ind messages. These three messages represent the three primitives defined in the 802 family of standards. The MAC-user will initiate packet transmission with the MA_req message enumerated below. This message will be sent to the mdata-id mailbox id previously obtained.

```
/*
 *      RQSTMSG (MA_DATA.request) STRUCTURE
 */
#define MA_req            0x2001  /* Message type */

#define RQSTMSG struct rqstmsg
struct rqstmsg {
        MSG      mh;              /* normal message header */
        char     LAC_chan;        /* LACS controller channel */
        char     frame_ctl;       /* frame control */
        long     *pkt_desc;       /* pointer to desc build area. */
        MBID     return_id;       /* return mailbox id */
        short    status;          /* return status field */
        short    type_fld;        /* type field or 802.3 length */
        LAN_ADR mac_DA;           /* 6 byte LAN Destination Addr */
};
```

Upon completion of the requested transmission the MAC will transform the users MA_req. message into an MA_conf message complete with status. The MA_conf message is enunmerated below. This message will be sent to the return mailbox id supplied in the MA_req message.

```
/*
 *      CONFMSG (MA_DATA.confirmation) STRUCTURE
 */
#define MA_conf           0x2002  /* error free .conf message type */
#define MA_cerr           0x2082  /* rejected request message type */

#define CONFMSG struct confmsg
struct confmsg {
        MSG      mh;              /* normal message header */
        char     LAC_chan;        /* LACS controller channel */
        char     frame_ctl;       /* frame control */
        long     *c_rsu;          /* pointer to desc build area. */
        MBID     return_id;       /* return mailbox id */
        short    status;          /* return status field */
        short    type_fld;        /* type field or 802.3 length */
        LAN_ADR mac_DA;           /* 6 byte LAN Destination Addr */
        short    csr0;            /* for debug, csr0 from 7990 */
};
```

```
/* .Confirm message return status error constants */
#define CER_TIMEOUT      0x80    /* no tx interrupt from chip */
#define CER_CHIPFAIL     0x81    /* chip specific failure */
#define CER_NOTSENT      0x82    /* could not get pkt onto medium */
#define CER_SHUTDOWN     0x83    /* MAC now out of service */

#define CER_ALLOC        0x90    /* could not alloc work space */
#define CER_HEAD         0x91    /* no space for mac header */
#define CER_LEN          0x92    /* packet too long for this mac */
#define CER_PAD          0x93    /* could not pad to min length */
#define CER_STRT         0x94    /* chip would not take start cmmnd
*/
```

When the MAC receives a packet from the medium destined for this module and without errors, it will compose an MA_ind message, enumerated below. This message will be sent to the muser_id mailbox id previously supplied by the user wishing to receive LAN packets.

```
/*
 *      INDMSG (MA_DATA.indication) MESSAGE STRUCTURE
 */
#define MA_ind           0x2003  /* error free .ind message type */
#define MA_ierr          0x2083  /* message type for bad .ind */

#define INDMSG struct indmsg
struct indmsg {
        MSG     mh;      /* normal message header */
        char    LAC_chan;        /* LACS controller channel */
        char    frame_ctl;       /* frame control */
        long    indmbz;          /* rsu and mbz */
        short   ret_code;        /* internel return code (status) */
        short   type_fld;        /* type or 802.3 length as rcvd */
        LAN_ADR mac_DA;          /* 6 byte LAN Destination Addr */
        LAN_ADR mac_SA;          /* 6 byte LAN Source Address */
        short   du_leng;         /* mac du length passed to LLC */
        short   csr0;            /* for debug, csr0 from 7990 */
};
```

In addition to the previously outlined user data interface there is also a System Management Interface. Systems management can request that actions take place and parameters be set and read. This interface uses 3 message types to exchange information as per the 802.1 standard. They are the LM_request, the LM_response and the LM_indicate. The LM request message will request that an action take place or a parameter be set or read. If the parameter is being read, the LM_response will carry the parameter value back to the systems manager. The LM_indicate message is sent from the MAC to systems management to indicate the occurrence of an event. These 3 message types have not been defined yet and consequently the code to handle them has not been put in place.

2.3 Initialization Requirements

The MAC lead task is spawned at start up time by the Kernel's initialization procedure. The following line in the SYSINIT table in the csl.c file will cause this to take place:

```
     [macinit, NULL, "MAC" 1, SUPER, NULL]
```

This line should always be present and it should come before any protocol initialization entries. The MAC lead task will determine how many daughter boards are present and what type they are. It will then spawn the appropriate local layer management process for the particular MAC daughter board.

For Ethernet, the layer management process will allocate space for an ecb for this port and spawn off the TX and RX processes. It will then create a data mailbox for itself, initialize the ecb and reset the Lance. Finally, it lowers its priority to four and awaits mailbox messages. The RX and TX modules will also create data mailboxes for themselves and obtain a pointer to their ecb. At this point this MAC is now in a quiescent state waiting for potential users to register themselves with the MAC to establish a message communication path.

A potential MAC-user must register itself with any MAC it wishes to use. This is accomplished by the user passing to MAC the mailbox id it wishes to receive MA_ind messages into and in turn, the MAC will return to it the data mailbox id to use when requesting transmission. The mechanism used to establish this link is MREGMSG structure enumerated below:

```
/*
 *      MREGMSG (MAC Registration) MESSAGE STRUCTURE
 */
#define MREGMSG struct mregmsg
struct mregmsg {
        MSG     mh;                 /* normal message header */
        short   status;             /* return status */
        MBID    return_id;          /* mboxid to return this message */
        MBID    muser_id;           /* MAC_user's MBID for .ind msgs */
        MBID    mdata_id;           /* MAC data MBID for data rqsts */
};
```

```
/*
 *      Message types for use with the MAREGMSG Message
 */
#define MU_REG_RQ        0x2010  /* MAC user reg msg type */
#define MU_REG_AK        0x2011  /* Ack for above msg type */

#define IR_REG_RQ        0x2012  /* Immediate Resp reg msg type */
#define IR_REG_AK        0x2013  /* Ack for above msg type */

#define SM_REG_RG        0x2014  /* Systems Mngmnt reg msg type */
#define SM_REG_AK        0x2015  /* Ack for above msg type */

#define LM_REG_RQ        0x2016  /* Layer Mngmnt reg msg type */
#define LM_REG_AK        0x2017  /* Ack for above msg type */
```

The registration message will be sent to the particular MAC's well-known layer management mailbox id. These well known names are "MAC_LM0", "MAC_LM1", "MAC_LM2", "MAC_LM3" and can be accessed by including an external declaration to the structure:

```
     char *mmwkn [ ];
```

2.4 Termination

This module and its associated sub-modules will always exist and should never terminate.

2.5 Environment

The MAC module operates as a specialized extension of the kernel operating system environment. The TX procedures should execute at a priority higher than all other protocol's TX procedures. The RX procedures should execute at a priority lower than all other protocol's RX procedures. The MAC modules must run with supervisor CPU privilege as they will disable and enable interrupts.

2.6 Timing and Size Requirements

Since the speed of this module is directly related to data throughput, it should be as fast as possible. The transmit module will append the MAC header and queue the packet for transmission as fast as possible. Other responsibilities related to the transmission will be done after queuing the message for transmission. The receive module will process the RXINT, strip the MAC header and send the packet off to the MAC-user (N+1 layer) as fast as possible thereby relinquishing control of the CPU. Other responsibilities related to packet reception will be handled when control is passed back to the module, presumably after the packet has left the highest on board layer. The size of the MAC lead task is approximately 1,980 bytes. The total size for the Ethernet specific MAC is approximately 15,880 bytes with the debug flag on.

2.7 Compilation/Assembly and Linking

The source module name for the MAC lead task is mac.c. The source module names for the Ethernet specific MAC code are: eth_lm.c, eth_rx.c and eth_tx.c. There are two header files containing constants and data structure definitions. They are called mac.h and ether.h. All modules will be developed in the /usr/dvlp/mac directory. Once they have become stable they will be moved by the unix system administrator to the /usr/rlse/mac directory. The file "makefile" controls compilation and includes all details for each particular compilation. Object files are created by typing "make lsts" in the /usr/dulp/mac directory. For users wishing to write code that interfaces with the MAC firmware they must include the following statement at the top of their source file:

```
     # include "../mac/mac.h"
```

When then the loadable bound unit is created, the user must link in the files /usr/dvlp/*.b or /usr/rlse/*.b in order to pick up the MAC object modules.

2.8 Testing Considerations

In the Ethernet specific MAC firmware provisions have been made to facilitate a software loopback. The user will compose an MA_req message with type TXLOOP. This message will cause the TX module to perform some basic error checks on the request, compose and send back a confirmation, and simulate an internal RXINT. The RXINT simulation will cause the RX module to compose an indication consisting of the data initially sent to the TX module and then to send this indication to the MAC-user's mailbox.

2.9 Documentation Considerations

Since this module is written in commented "C" and intended for internal use, the code itself in conjunction with this specification will serve as documentation.

2.10 Operating Procedures

None

2.11 Error Messages

The only error message that will be returned to the user will be in the form of a status word contained in the MA_conf message. These are enumerated below.

```
/* .Confirm message return status error constants */
#define CER_TIMEOUT      0x80    /* no tx interrupt from chip */
#define CER_CHIPFAIL     0x81    /* chip specific failure */
#define CER_NOTSENT      0x82    /* could not get msg onto medium */
#define CER_SHUTDOWN     0x83    /* MAC out of service, shutdown */

#define CER_ALLOC        0x90    /* could not alloc work space */
#define CER_HEAD         0x91    /* no space for mac header */
#define CER_LEN          0x92    /* packet too long for this mac */
#define CER_PAD          0x93    /* could not pad to min length */
#define CER_STRT         0x94    /* chip would not take start cmmnd
*/
```

Other possible errors will be handled by the protocol as per the 802 family of standards. When appropriate an error counter will be incremented. These error counters are readable via the systems management interface.

3.0 INTERNAL SPECIFICATIONS

3.1 Overview

The MAC firmware has been designed to handle up to four MACs of the same or different type. The MAC function itself will vary only minimally from access method to access method and will not vary at all with respect to the MAC-user interface. At the current time only the Ethernet MAC is supported, however, the concept will remain the same as more MAC specific handlers are generated to control other access method daughter boards.

3.2 Subcomponent Description

3.2.1 macinit()

The macinit task is the lead task spawned by the kernel and is always the first to run. It will determine which ports are used and what type of MAC is present on each of the used ports. It will then spawn a layer management task for the specific MAC type. This ends its involvement into the particulars of that MAC.

The only other critical task that this module performs is to register a generic MAC interrupt handler with the kernel. Due to hardware design considerations there is only one interrupt from all the daughter boards, thus there can be only one interrupt handler. This handler will determine which of the four daughter boards has generated an interrupt. It will then perform a "C" subroutine call to the particular interrupt handler responsible for the details of this particular MAC chip set. The method by which this linkage occurs is contained in the only data structure created by macinit. The structure is called the MAC_HEAD global data structure. It contains a pointer to the MAC's unique data structure and a pointer to the MAC specific interrupt handler for each of the four possible ports. These pointers must be placed in the data structure as part of the MAC-specific initialization procedure. It is the only means by which the generic MAC can make use of code tailored to a specific MAC.

3.2.2 eth_lm()

This is the local layer manager for the Ethernet specific MAC daughter board. It will allocate space for the ecb from PRIVATE memory and place a pointer to it in the MAC_HEAD structure indexed by port number. It will then place a pointer to the 7990-Ethernet interrupt handler in the MAC_HEAD structure, again indexed by port number. Next it will spawn a copy of the TX and RX modules for this port and create a data mailbox for itself. Finally it will initialize the ecb structure.

As part of the ecb initialization, SHARED memory must be allocated for the LANCE. The LANCE will use this memory for its initialization block, receive descriptor ring and transmit descriptor ring. When all initialization is complete the layer manager will lower its running priority and wait on incoming messages.

The messages that this module will receive are: MAC-user registration requests, system management exchange requests, non-RX/non-TX Lance interrupts (error condition) and reset requests generated by the TX or RX modules upon detecting a fatal error related to quirks in Lance operation requiring the chip to be reset.

3.2.3 eth_rx()

The Ethernet RX module is responsible for processing incoming data packets from the LAN. It is a shared stack process due to the nature of the function it performs. It makes use of the ecb structure for this port, a pointer to which is obtained from the MAC_HEAD structure indexed by port number. After performing some basic initialization which includes allocating a data mailbox, it waits for incoming message.

The message types that this module expects to receive are: RXHALT and RXRESUME generated by the local layer management as part of its reset procedure, LOOPBACK which is a TX generated receive interrupt simulation, and most importantly, a genuine RXINT sent to it by the Ethernet specific interrupt handler.

Upon receiving an RXINT, the RX module will inspect the Lance's status register supplied in the message for errors. It pulls the used receive descriptors from the Lance's receive message descriptor ring and temporarily stores them in the ecb. It then replaces the used descriptors (an implicitly the data buffers) with fresh pre-allocated descriptors. It now inspects the descriptor status words for errors. If any errors are detected the packet is discarded. If the packet is error free the RX module will transform the RXINT message into an MA_ind message. It will place in the message a pointer to the (Kernel) data buffer descriptor, the packets destination and source addresses as well as the Ethernet type field (802.3 length field). Next it strips or "unprepends" the MAC header from the packet, shortens or "unappends" the buffer to the actual packet size and places the size in the indication message. At this point it sends the MA_ind message to the mailbox id which the MAC-user has registered with the local layer manager. Lastly it preallocates new buffers to replace the ones just used.

The RX module contains 4 subroutines to simplify the code flow. They are: "retrieve_rxdesc()" used to move valid descriptors from the Lance's receive message descriptor ring to the ecb. It takes a pointer to the ecb as input and returns the number of desciptors moved; "replenish_rxdesc()" used to replace the most recently used descriptors in the Lance's receive message descriptor ring with fresh descriptors preallocated and stored in the ecb. It takes a pointer to the ecb and the number of descriptors to replenish as input and returns nothing; "allocate_rxdesc()" used to preallocate buffer space and generate fresh descriptors. It gets the buffer space via the getbuf kernel call and stores the fresh descriptors in the ecb. It takes a pointer to the ecb and the number of descriptors to allocate as input and returns nothing; and "reallocate_rxdesc()" used to recycle descriptors that pointed to a receive packet that was in error. This routine saves a freebuf and getbuf kernel call. It takes a pointer to the ecb and the number of descriptors to replenish as input and returns nothing.

3.2.4 eth_tx()

The Ethernet TX module is responsible for processing outgoing data packets on behalf of the MAC-user. It also contains the Ethernet specific interrupt handler. It is a shared stack process due to the nature of the function it performs. It makes use of the ecb structure for this port, a pointer to which is obtained from the MAC-HEAD data structure indexed by port number. After performing some basic initialization which includes allocating a data mailbox and allocating a permanent alarm message, it awaits incoming messages.

The message types this module expects to receive are: TXHALT and TXRESUME generated by the local layer management as part of its reset procedure, LOOPBACK which is a MAC-user generated debug function, MA_req which is the normal means by which the MAC-user initiates a transmission, TXINT which is sent to the TX module by the ethernet specific interrupt handler in response to a Lance transmit interrupt and finally ALARM which is sent to it by the kernel´s alarm facility to indicate that a TXINT is overdue.

Upon receiving an MA_req from a MAC-user, the TX module will allocate PRIVATE memory for building the transmit descriptors. It will store the pointer to this area in the MA_req message. It will grow or "prepend" space onto the beginning of the data-unit for the MAC header and then install the destination address, source address and Ethernet type field (802.3 length field) from information contained in the MA_req message. It will pad the data-unit out to 64 bytes total using nulls if the length was less than 64 bytes. It then creates a Lance descriptor for each segment of the data-unit indicated by the number of kernel buffer descriptors. It next queues this MA_req message (complete with descriptors) at the end of a private mailbox-like data structure in the ecb called the pending queue. Finally, it checks the activity-flag for this port.

If the activity flag is false, the descriptors will be copied into the Lance´s transmit message descriptor ring, the activity flag set true, a timeout alarm set, and finally, the Lance is commanded to start transmission. If the activity flag were true the TX module would exit back to the kernel where it awaits the next mailbox message. Thus message transmission is fully interrupt driven once messages are queued.

Upon receiving a TXINT from the Ethernet specific interrupt handler, the TX module will first stop the timeout alarm and then process the interrupt. It will inspect the Lance's status register supplied in the TXINT message for errors. It will take the message from the top of the pending queue which was the original MA_req for this packet and transform it into an MA_conf message. It then retrieves the descriptors from the Lance's transmit message descriptor ring and temporarily stores them in the ecb for inspection. It now inspects the descriptor status words for error conditions, any of which would indicate that the transmission failed. The status information is coded and placed in the confirmation message. The confirmation message is finally sent back to the return mailbox id called out in the original MA_req message. Lastly, it inspects the pending queue of MA_reg messages previously queued by the MA_req code handler. If the queue is empty, the activity flag is set false. If the queue is not empty, the top entry will be started. The pre-formed descriptors will be copied into the Lance's transmit message descriptor ring, a timeout alarm set and the Lance commanded to start transmission.

Upon receiving an alarm message from the kernel's alarm facility the TX module will assume the worst. It will locate the MA_req at the top of the pending queue, return the memory containing the descriptors back to the free pool and transform the MA_req into a confirmation. It will set the appropriate status value and return the message to the indicated return mailbox id. Finally, it causes the Lance to be reset by making a TXREST request on the local layer manager. The layer manager will request that the RX module halt and cleanup. When the RX module has halted the layer manager will perform a drastic reset. The drastic reset is equivalent to initialization with the exception that no new memory is allocated for an ecb or Lance control blocks and the error statistics are not reset.

The TX module also contains four subroutines to simplify the code flow. They are: "retrieve_txdesc()" used to recover the used descriptors from the Lance's transmit message descriptor ring and place them in the ecb for inspection. It takes a pointer to the ecb as input and returns the number of descriptors moved; "fill_desc()" used to create a Lance transmit descriptor from the information contained in the kernel buffer descriptor. It takes a pointer to the kernel buffer descriptor and a pointer to the build location as input and returns nothing; "padpkt()" used to extend data-units to the legal minimum size for ethernet (64 bytes). It pads with nulls (0x00). It takes a pointer to the kernel buffer descriptor as input and returns -1 if the operation could not be accomplished, 0 otherwise; "start_xmit()" used to command the Lance to begin transmitting a packet. It will set the timeout alarm, copy the indicated preformed descriptors into the Lance's transmit message descriptor ring, command the Lance to begin transmission and set the activity flag. It takes a pointer to the ecb for this port as input and returns -1 if it could not perform the function, 0 otherwise.

3.2.5 int_ethernet()

The int_ethernet routine is the ethernet specific interrupt handler bound to the generic MAC interrupt handler by virtue of a pointer to it installed in the MAC_HEAD data structure for this port. This routine is a subroutine called by the generic MAC interrupt handler and together both routines run at interrupt level. The source for this module is contained in the eth_tx.c source file.

This routine takes the port number as input and returns nothing. It makes use of the ecb structure for this port, a pointer to which is obtained from the MAC_HEAD structure indexed by port number. It will read and reset the Lance's control and status register (csr0), and reset the hardware interrupt. It allocates PRIVATE memory to compose an interrupt message. The message will contain the port number and the value of the Lance's csr0 register. It then inspects the status register for a receive or transmit interrupt and sends the interrupt message to either the RX or TX process respectively. If neither type interrupt is indicated in the status register the message is sent to the local layer management for this port where a decision is made as to the cause of the interrupt.

3.3 Future Development and Maintenance

Future development will be necessary as new MAC specific daughter boards are designed. The existing MAC firmware modules have been designed with this in mind. The ethernet specific modules should act as a template for future development. In general the programmer should design a layer manager and a TX and RX module which would be spawned by the layer manager. This will allow a certain degree of freedom in the specifics of spawning the TX and RX modules. Any data structures used by the three modules must be gotten from free memory (PRIVATE). This is due to the fact that from zero to three copies of the modules may be spawned, each needing a unique data structure.

Any new modules must use the MAC-user message structure called out in the mac.h header file. Finally, the programmer must go in and edit the macinit routine in the mac.c source file to include the line which will spawn the MAC specific layer manager.

As part of the layer manager, the programmer must install pointers to his/her MAC specific data structure and MAC specific interrupt handler in the MAC_HEAD data structure. After that, all he/she has to do is code the specifics necessary to handle the particular MAC (if it has any peculiarities ie. 802.4 immediate response) and the chip set and hardware implementing the access method.

4 PROCEDURAL DESIGN

4.1 Generic MAC Subsytem Initialization, macinit()

```
register (generic interrupt);
for (i = 0; i <= 3; i++)
     get daughter board id (i);

     switch (daughter board id)
     case (ETHERNET):
          procreate (layer manager);
          register (default mailbox id);
          prorun (layer manager);
          break;

     case (TOKENBUS):
          /* TBD */
          break;

     case (TOKENRING):
          /* TBD */
          break;

     case (STARLAN):
          /* TBD */
          break;

lower running priority (4);
```

4.2 Ethernet Layer Managment procedure, elminit()

```
MAC_HEAD.int_proto[port] = int_ethernet;
ecb = allocate (sizeof (ecb), PRIVATE);
MAC_HEAD.data[port] = ecb;
procreate (etxinit)           /* start the TX procedure */
prorun (etxinit)
procreate (erxinit)           /* start the RX procedure */
prorun (erxinit)
create (data mailbox);
ecb_initialize (ecb);
reset_status (ecb);
lower running priority (4);

for EVER
    breceive (message, mboxid);    /* await message */

    switch (message.type)
    case (MU_REG_RQ):
        ecb.mu_did = message.muser_id;
        message.mdata_id = ecb.tx_did;
        sendmsg (message, message.return_id);
        break;

    case (LM_REG_RQ):
        ecb.event_did = message.muser_id;
        message.mdata_id = ecb.lm_did;
        sendmsg (message, message.return_id);
        break;

    case (LMINT):
        /* TBD */
        break

    case (RESET):
        if (RXRESET)
            sendmsg (TXHALT, ecb.tx_cid);
        else
            sendmsg (RXHALT, ecb.rx_cid);
        break;

    case (HALTED):
        ecb_reset (ecb);
        load_block (ecb);
        chip_reset (ecb);
        sendmsg (RESUME, ecb.tx_cid);
        sendmsg (RESUME, ecb.rx_cid);
        break;
```

4.3 Ethernet Transmit Procedure, etxmain()

```
etxmain (message, mboxid);

switch (message.type)
case (TXHALT):
     tx_cleanup (ecb);
     sendmsg (TXHALTED, ecb.lm_cid);
     break;

case (ALARM):
     conf_msg = top_of_queue;
     mfree (desc_build_area);
     conf_msg.status = CER_TIMEOUT;
     sendmsg (conf_msg, conf_msg.return-id);
     tx_cleanup (ecb);
     message = allocate (48, PRIVATE);
     message.type = TXRESET;
     sendmsg (message, ecb.lm_cid);
     break;

case (MA_req):
     tx_msg = MA_req message;       /* reuse message */
     allocate (48, PRIVATE);        /* to build descriptors */
     bd = tx_msg.m_bufdes;
     unprependbuf (bd, 14);         /* make room for header */
     move DA into header;
     move SA into header;
     if (packet_length > 1500)
          /* packet too big */
          tx_error (CER_LEN);
     if (packet_length < 64)
          /* packet too small */
          padpkt ();
     if (packet_type == 0)          /* 802.3 if 0 */
          type = length;
     frame_type_field = type;
     for (; bd != NULL; bd = bd->bd_next)
          fill_desc (bd);     /* create a desc for each bd */
     set STP in first descriptor;
     set OWN in all descriptors;
     set ENP in last descriptor;
     disable interrupts;
     queue MA_reg at end of pending queue;
     if (activity_flag)
          enable interrupts;
     else
          enable interrupts;
          start_xmit(ecb);
     break;
```

```
case (TXINT):
     stop_alarm();
     conf_msg = top_of_queue;
     mfree (descriptor_build_area);
     num_descriptors = retrieve_rxdesc (ecb);
     if (first descriptor has bad status)
          sendmsg (conf_msg, conf_msg.return_id);
          sendmsg (TXRESET, ecb.lm_cid);
     if (subsequent descriptor has bad status)
          status = CER_NOTSENT;
     if (status != 0)
          conf_msg.type = MA_cerr;
     else
          conf_msg.type = MA_conf;
     sendmsg (conf_msg, conf_msg.return_id);
     if (pending_queue not empty)
          start_xmit (ecb);
     else
          activity_flag = false;
     break;
```

4.4 Ethernet Receive Procedure, erxmain()

```
erxmain (message, mboxid);

switch (message.type)
case (RXHALT):
     rx_cleanup (ecb);
     sendmsg (RXHALTED, ecb.lm_cid);
     break;

case (RXLOOP):
     bd = message.m_bufdesc;
     move DA into ind_msg;
     move SA into ind_msg;
     unprependbuf (bd, 14)        /* strip mac header */
     ind_msg.type = MA_ind;
     sendmsg (ind_msg, ecb.mu_did);
     break;

case (RXINT):
     if (csr0 indicates ERROR)
          if (csr0 indicates "MERR" or "MISS")
               rx_cleanup (ecb);
               sendmsg (RXREST, ecb.lm_cid);
     num_desc = retrieve_rxdesc (ecb);
     replenish_rxdesc (ecb, num_desc);
     for (; num_desc > 0; num_desc--)
          check all descriptors for errors;
          if (bad status)
               update ring pointers;
               reallocate_rxdesc (ecb, num_desc);
               break;
     get bd from retrieve descriptor;
     ind_msg = RXINT message           /* reuse message */
     move DA into ind_msg;
     move SA into ind_msg;
     move type_field to ind_msg;
     unprependbuf (bd, 14);            /* strip header */
     unapendbuf (bd, (buffer_length - packet_length));
     ind_msg.type = MA_ind;
     sendmsg (ind_msg, ecb.mu_did);
     update receive ring pointers;
     allocate_rxdesc (ecb, num_desc);
     break;
```

5 ISSUES

Removing restrictions imposed on the ethernet firmware by problems in the Lance is of great concern. The fact that these problems have been remedied by AMD must first be verified before the restrictions are removed. If the conditions go unchecked they could be very hard to track down.

A strict requirement should be established to only allow ethernet version 2.0 or 802.3 tranceivers to be used with the LACS. Early ethernet tranceivers do not have heartbeat and this fact is reported by the Lance as a "late-collision" error. If this cannot be done, code must be added to ignore this error. This will add extra overhead on every packet transmitted as it causes the Lance´s general error bit to be asserted causing the ethernet firmware to immediately take a more time consuming error handling path.

The requirement for the LACS to use ethernet 2.0 or 802.3 tranceivers pertains only to the LAC´s connection to the network. It has no bearing on any other nodes on the network.

The layer management to systems management interface has not yet been fully defined. A requirement from the systems management designer must be forthcoming before code can be generated to handle systems management requests.

The in-line T&V must be designed and integrated into the current ethernet firmware. The T&V designers have been made aware of the most optimum means by which to realize this integration. However, as with anything, debug and checkout could reveal some surprises.

A substantial time savings may be realized if the confirmation message theory of operation is deleted from the MAC firmware. There have been some interesting discussions on this topic with Ron Dhondy. He should be consulted for further details. Basically, there seems to be no good reason to have such a mechanism in an implementation that uses type I LLC services under a class 4 transport layer. If the MAC-user is operating a connectionless unidata service, it will by definition ignore the contents of an MA.confirmation. Furthermore, even with a connection oriented type II LLC service, the confirmation it is looking for will be coming from its remote peer entity. The MAC´s confirmation is only signifying that the packet left the local node

Finally there is the issue of future MAC-specific hardware and the design of firmware to control it. If one is careful to observe the hooks and handles purposely designed in for this purpose it should be relatively easy for the various MAC-specific modules to co-exist.