NASA CR-172032

Media Independent Interface
Interface Control Document

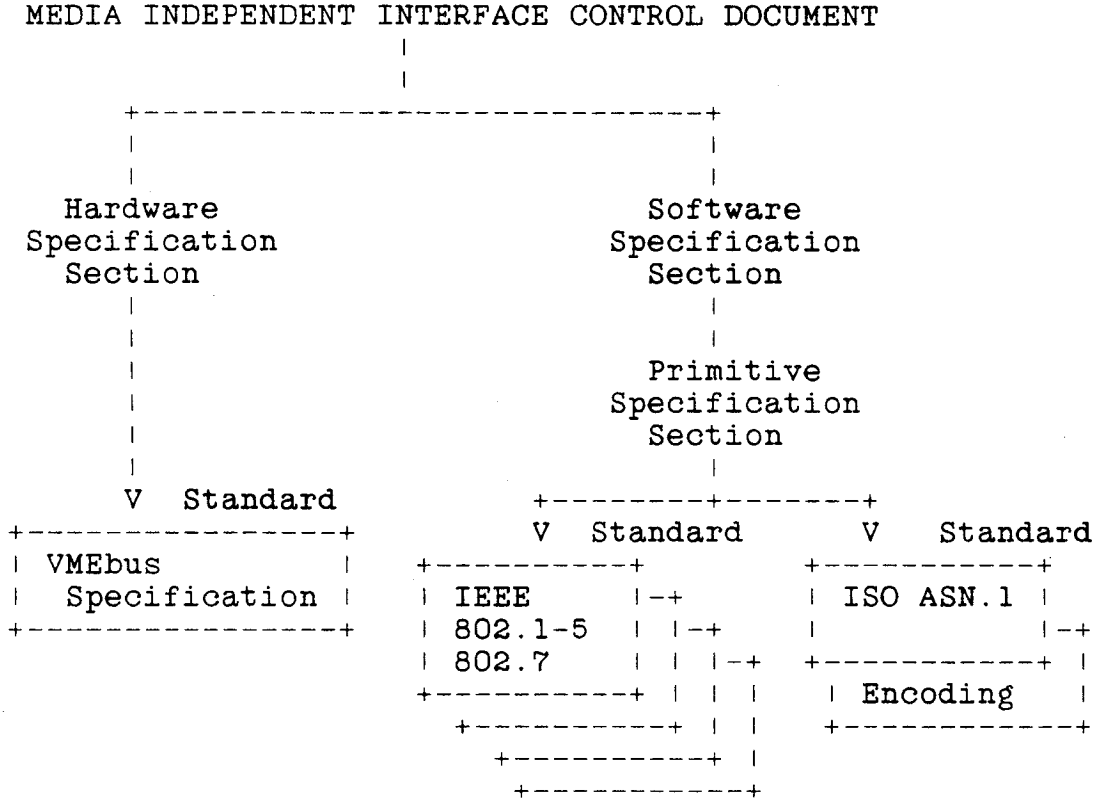
MS 2-2-5250
SPERRY SPACE SYSTEMS
P.O. BOX 52199
PHOENIX, ARIZONA
85072-2199

CONTENTS

1.2 SCOPE

This document specifies a Media Independent Interface. This interface uses standards current in the industry. The industry standards were generated at different times and have differing applications. This has lead to a convoluted and poorly defined set of interface primitives. They lack a cohesive binding to each other. In addition the standards do not describe the physical aspects of these standards which would allow different implementations of them to operate in concert. The MII ICD binds these standards by integrating mechanical, electrical, and functional interface standards within this document.

The MII is described in hierarchical fashion. At the base are IEEE/ISO documents (standards) which describe the functionality of the software modules or layers and their interconnection. These documents describe primitives which are to transcend the MII. They do not describe the method by which layers communicate or the exact language in which they speak. In addition different specs sometimes disagree in the structure of the same primitives. These standards specify the logical interface.



These logical specifications are further defined in this ICD with the use of a canonical language. This canonical language along with the physical and electrical specifications called out in this document provide the final binding required to make a working interface.

The structure of this standard is pictured above.

1.3 ABBREVIATIONS

ASN.1 - Abstract Syntax Notation One
CPU - Central Processing Unit
DIS - Draft International Standard
ICD - Interface Control Document
ICI - Interface Control Information
IEEE - Institute of Electrical Electronics Engineers
ISO - Internation Standards Organization
LAN - Local Area Network
LLC - Logical Link Control
MAC - Media Access Control
MII - Media Independent Interface
OSI - Open Systems Interconnection
PDU - Protocol Data Unit
SDU - Service Data Unit
SM - Station Management

1.4 DOCUMENTS

ANSI X3T9/84-100 Fiber Distributed Data Interface (FDDI)
IEEE 802.1 (still draft) Network management
IEEE 802.2 or ISO 8802/2 Logical Link Control (LLC)
IEEE 802.3 or ISO 8802/3 Carrier Sense Media Access (CSMA/CD)
IEEE 802.4 or ISO 8802/4 Token Bus
IEEE 802.5 or ISO 8802/5 Token Ring
IEEE 802.7 or ISO 8802/7 Slotted Ring
IEEE P1014/D1.0 VMEbus/ Signetics VMXbus
ISO DIS 8824 Specification of Abstract Syntax Notation
ISO DIS 8825 Basic Encoding for Abstract Syntax Notation
ISO 7498 ISO OSI Basic Reference Model
ISO 7498 DAD1 connectionless Data Transmission
ISO DP 7498/4 Management Framework
Sperry Report 2055-04 thru 2055-8

2 GENERAL

The intent of the MII is to provide a universal interface to one or more MACs for the Logical Link Controller and Station Manager. This interface includes both a standardized electrical and mechanical interface and a standardized functional specification which define the services expected from the MAC.

3 ARCHITECTURE

3.1 INTRODUCTION

Communication between computers has always been difficult when a common design was not used for all the computers. It has become necessary to provide a means whereby dissimilar computers communicate. To supply that need IEEE created a number of standards which could be built by different designers and yet could pass information between them. The information passed between them was also slightly dissimilar in content and structure and again there was a need for another standard. The International Standards Organization (ISO) built a 7 layer model by which the control of information passing could be standardized. The model adopted the IEEE standards. These IEEE standards are the first two layers of a 7 layer implementation where the lowest layer is the physical connection between the computers.

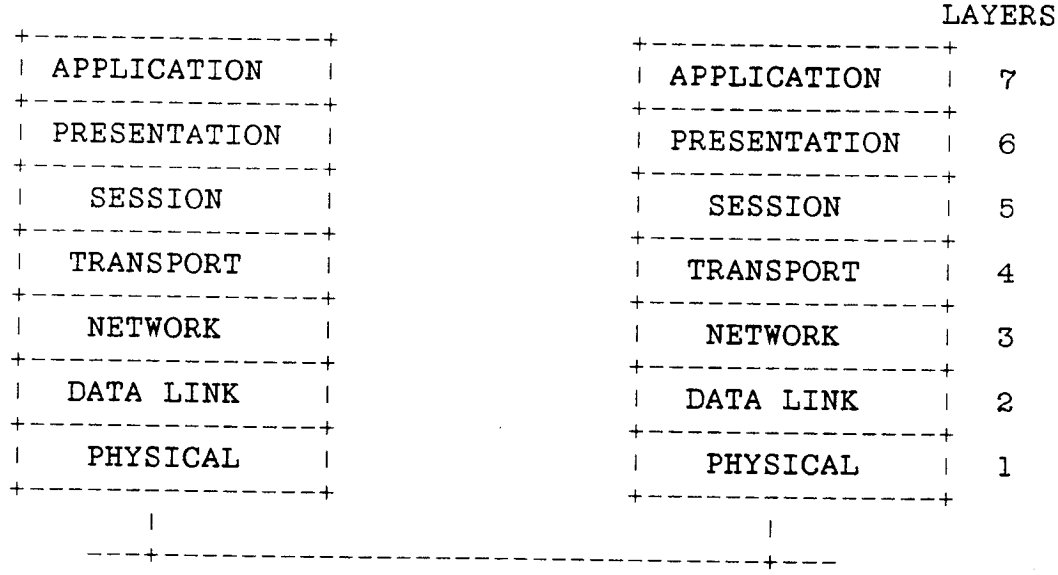


Figure 1.0 The ISO OSI model

The IEEE standards used in this document are used in the lower two layers of the ISO OSI model above. These layers are sub-divided by the IEEE standards.

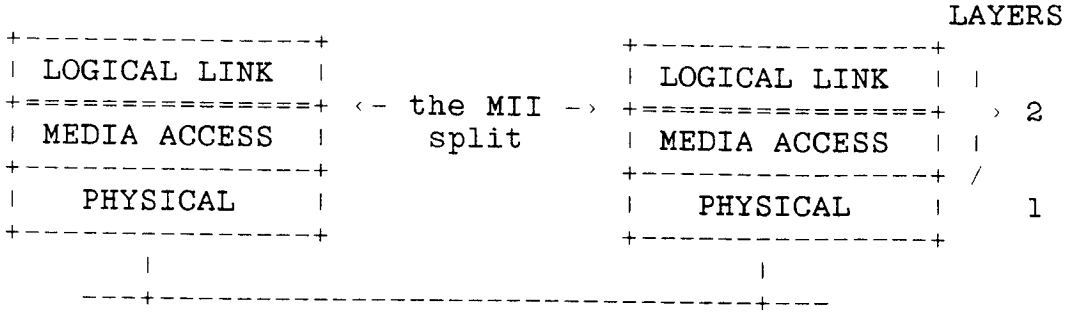


Figure 2.0 The IEEE Model

The IEEE committee realized the need to be able to use a common Logical Link Control for several versions of the Media Access and Physical layers. The IEEE standards provide compatibility only on a logical level. They do not describe the implementation. There are inconsistencies between MACs. Some MACs provide more services than others. This makes standardized interchanges difficult even on a purely logical level. Finally the initialization and maintenance of the MACs are considerably different which complicates the matter further.

The purpose of the MII is to allow standard interchanges between the MACs and the LLC. The interchanges requires a Station Manager which can effectively operate the entire selection of MACs. Fortunately they're all similar in their management and all of them tend to operate without a lot of external intervention.

3.2 HOW THE ARCHITECTURE WORKS

Two kinds of information must be passed in order to move data between the LLC and MAC; the data to be shipped and shipping instructions.

The instruction part of this information is called Interface Control Information (ICI) and is layer to layer instructions. It includes instructions as to where this data can be found, its sender and receiver,. and its quantity. The MII instructions are limited to a standard set of primitives (MA_DATA.Request, MA_DATA.Confirm, MA_DATA.Indicate). These primitives and their effect are defined in the IEEE standard. The specification is a logical one and does not attempt to provide guidance as to its implementation. In this ICD, the primitives have been re-written in a standard syntax and encoding for implementation clairification. Where primitive parameter differences between MAC standards occur, the syntax provides the layers an ability to default missing parameters and ignore extra ones. If a LLC has the capability to use all the services of a MAC, then the interface will allow it to do so. If not, then it must depend on the non-optional primitive parameters and rely on the MAC to default any additional ones.

The Data information to be shipped is called the Protocol Data Unit (PDU) and contains the original message and peer to peer layer messages. Both the data and the peer layer information has no direct effect on the operation of the MII. It simply must be passed intact across it.

To provide a physical means to implement the above activity, an architecture has been selected which can be supported with an existing standardized bus. The architecture is as follows;

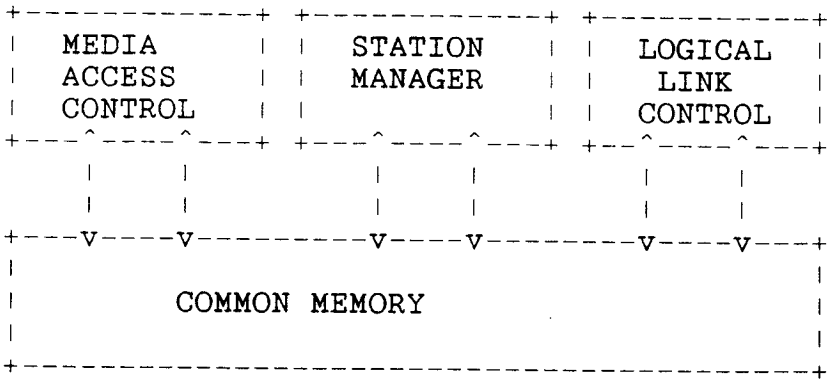


Figure 3.0 The Logical Model

This model does not restrict the MAC, SM and LLC except that they must be able to access common memory in order to communicate. Data flow is always directed through this common memory element. If an implementation of the MAC has its communication channel located on-board as locally and globally decoded dual ported memory then it will have a speed advantage. It will be able to read and write to this part of the common memory with a local access and detect global reads and writes via local interrupts. This eliminates one of two parties from having to arbritrate for the bus.

Each sending entity knows the address of the receiving entity channel. Any time a entity wishes to pass information to another entity, it simply writes the address of the information into the receiver's channel (address). If the receiving channel's memory location is decoded on board, the receiving entity's physical device and the pointer will be read with a local access.

Local decoding of the receive channel is not required. The channels physical implementation is two valid memory locations in common memory, which could be in regular memory (i.e. as dual port memory or mapped to read/write registers). Any standard CPU card with the ability to read and write global memory, will be able to implement a receiving channel.

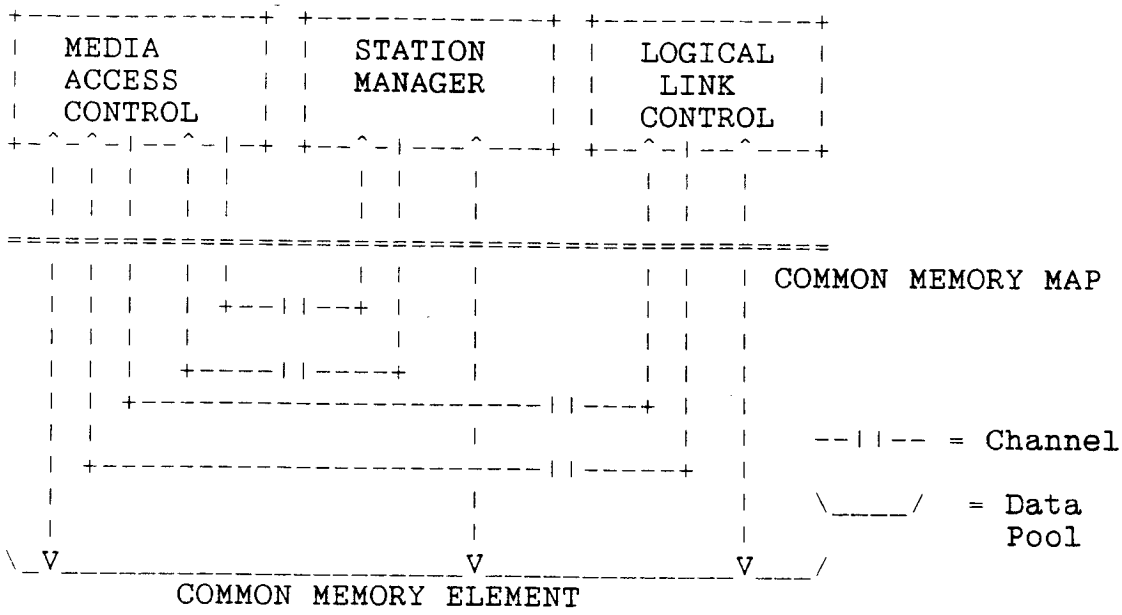


Figure 4.0 The Data Flow Model

The MII data flow diagram shows that the MII is actually a bus with channels of communication. The channels are two sequential address locations mapped in the common (sometimes called global) memory map. These address locations provide the same hardware services that are found in standard VME memory (i.e. long addressing, DTACK, Buserror, etc). The first location is tested to see if the channel is busy. If not then the address of data to be passed is written to the second memory location. The entity receiving the date uses the address as a pointer to the information stored in global memory and subsequently resets the channel busy location. The busy location is known as the channel semaphore.

The address passed via the channels is the Interface Control Information (ICI). The ICI contains information as to the location of Protocol Data Units (PDU). The ICI primitives are described in the IEEE specifications and their syntax and encoding structure is described in this document.

This MII architecture is implemented on a standard bus which uses a multi-master architecture. It has various options, all of which are allowable in the MII. The system designer need only be concerned with the performance resulting from his choices of entity designs and bus options. Choices such as the size of the common memory available to the MAC for PDUs are not restricted by the MII. The designer is free to choose between MACs with any amount of on board buffering.

There may be MAC designs which have no on-board buffers and require the card to be set to preempt bus users when packets arrive.

The only restrictions imposed by this MII ICD are the use of VMEbus, enough common memory between users to implement the channels and store ICIs and PDUs, and the use of the standard primitive syntax as defined in this document.

3.3 MII HIGHLIGHTS

The MII solutions given in this ICD are the result of a unique blend of advanced technology integrated with the ever expanding world of standards. This integration is made possible by the careful selection of the hardware platform and specialized development of software interface technology. The MII ICD is the product of a continuous effort in the advancement of state of the art communications. The feasibility of the MII ICD was demonstrated by two fiber optics developments which have a media bandwidth that exceeds most of current industry implementations by a factor of ten.

Even more important is the MII ICDs unique ability to grow with technology. Its method of information exchange was explicitly developed and exercised to establish a decisive interface. This interface was adapted to two disimilar Media Access Controllers running at 100 M-bit/s in order to illustrate its capacity to meet and exceed the needs of custom and standard users alike. As the communication industry realizes ever increasing sophistication in communication functionality, the MII ICD will continue to demonstrate its flexibility.

The MII;

- uses IEEE, ISO and VMEbus Standards,
- is useable for both ground and flight systems,
- provides complete interchangeability between MACs,
- supports bridges, routers, and gateways,
- allows changes in the LLC, SM, or MAC,
- is expandable yet retains backwards compatibility,
- does not interfere with upper layer software,
- is standard mechanically, electrically, and functionally,
- and defines mechanical, electrical and functional aspects.

3.4 SYSTEM REQUIREMENTS

A system supporting a MII must contain at least one VMEbus, enough globally addressable memory and bus bandwidth to allow operation of the LLC, SM, and MAC entities. At minimum, the bus must have enough slots and capacity to provide Master capacity for the VME card(s) containing the MAC, SM and LLC entities.

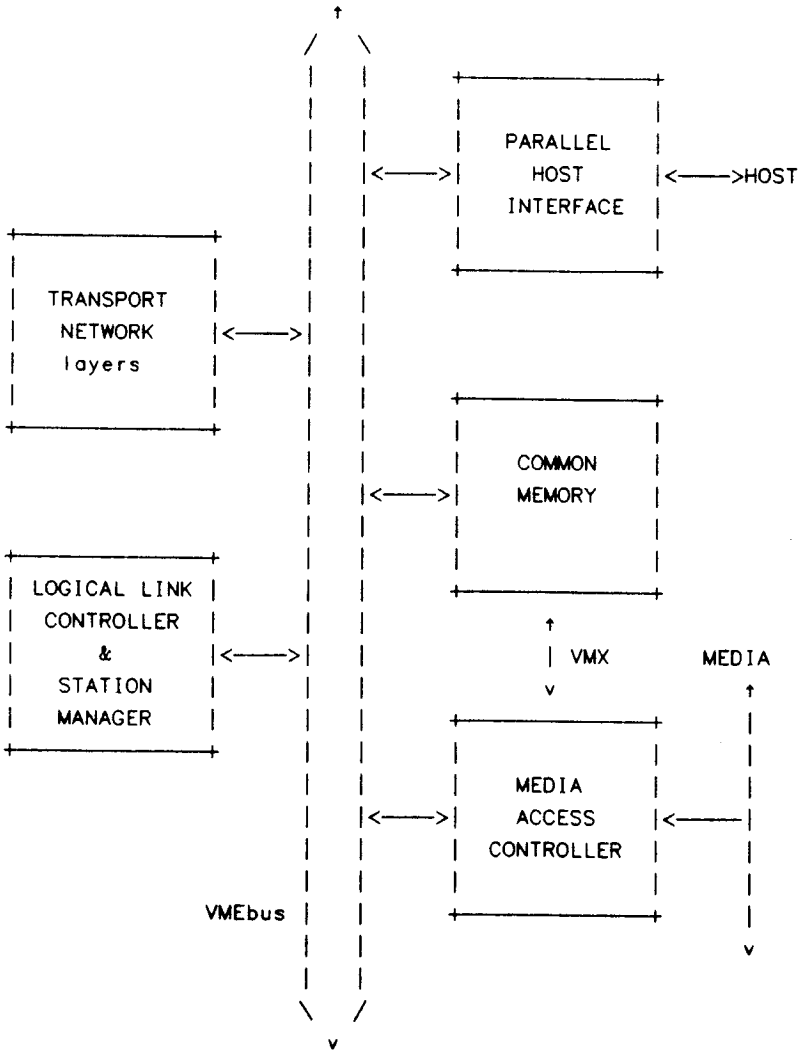


Figure 5.0 Example Configuration

Selection of VMEbus options such as round robin arbitration or priority arbitration is not limited by the MII and is up to the system designer. These options are all allowable providing all devices are consistent with the VMEbus specification.

3.5 INTERFACE CONTROL SCOPE

The MII ICD was design to allow multiple cards for each entity (MAC, LLC, SM) and multiple entities (LLC and SM) per card. For example if a MAC design requires a large memory, a second card can be used to support it. The addressing of that card would have to be compatible with the whole system. If mapping or speed is a problem, then the VMX bus might be used. There are no restrictions along these lines.

The number of entities across the MII is not restricted. The ability of the entities to respond or even recognize different combinations of entities is left open for the implementor. For instance the MII was designed to allow multiple MAC entities per LLC. The interface will support two or more LLCs with matching MACs. Multiple LLCs or SMs per MAC is supported. The System designer is responsible for insuring that the devices are compatible and that combinations and quantity are appropriate for the design. The MII does not limit the design of any of the entities except to insure that they can communicate in a known manner. The MII contains an inherent flow control which will eliminate overrun of MII interfaces. The sustained data rate across the MII is limited only by the bandwidth of the VMEbus and the size of memory allowed on the VMEbus. Speed of the entities and their ability to effectively use the bus is considered a system design issue beyond the scope of the MII requirements.

3.6 MECHANICAL AND ELECTRICAL

The MII relies on the VMEbus and VMX specification to provide the electrical and mechanical platform on which MII resides. VMX interface is allowed to enhance system performance. It is considered an extension of the VMEbus and the MII devices must conform to the same MII requirements whether a VME or VMX operation is being performed.

The following is not specified or limited by the MII;

- Bus addressing and priority,
- physical slot location,
- the number of physical cards,
- the division of functionality between cards,
- the amount of memory,
- the use of DMA processors and support devices.

3.7 SOFTWARE

The software aspect of the MII is a virtual interface. The MII users are configured to recognize other MII entities by addresses only. Entities using the MII are made aware of the other devices addresses either by mechanical means (dip switches, proms, etc) or by software interaction.

The Station Managers role is to provide information to the LLC and MAC which will allow operation in the system environment. In a typical software environment memory allocation is controlled by the operation system. The memory used by MII users is allocated by the Station Manager and given to the users via the MII ICD method. This avoids standardizing an operating system just to assure correct memory usage. In addition to common memory allocation to MII users, the Station Manager provides addresses to the user entities which indicate channel locations. Each MII user entity has the capacity to receive initialization information upon startup according to the MII prescribed method.

The MII transfers information by passing pointers to data structures located in common memory. The MAC has two receive channels to receive the pointer addresses from the Station Manager and LLC. The LLC and SM have a single receive channel each to receive pointer addresses from the MAC. Receiving channels beyond these required ones are not defined by the MII ICD and are beyond its scope.

Transmitting occurs only after a non-interrupting test and set is made of the channel semaphore location and ownership to use the channel is established. The entire transmission involves a write of a single long word address into the memory location following the semaphore (semaphore_address + 1 (long address)). These locations are shown in figure 7.0. See the VMEbus Specifications for address length definitions. The logical flow of access to a receive channel is diagramed as follows;

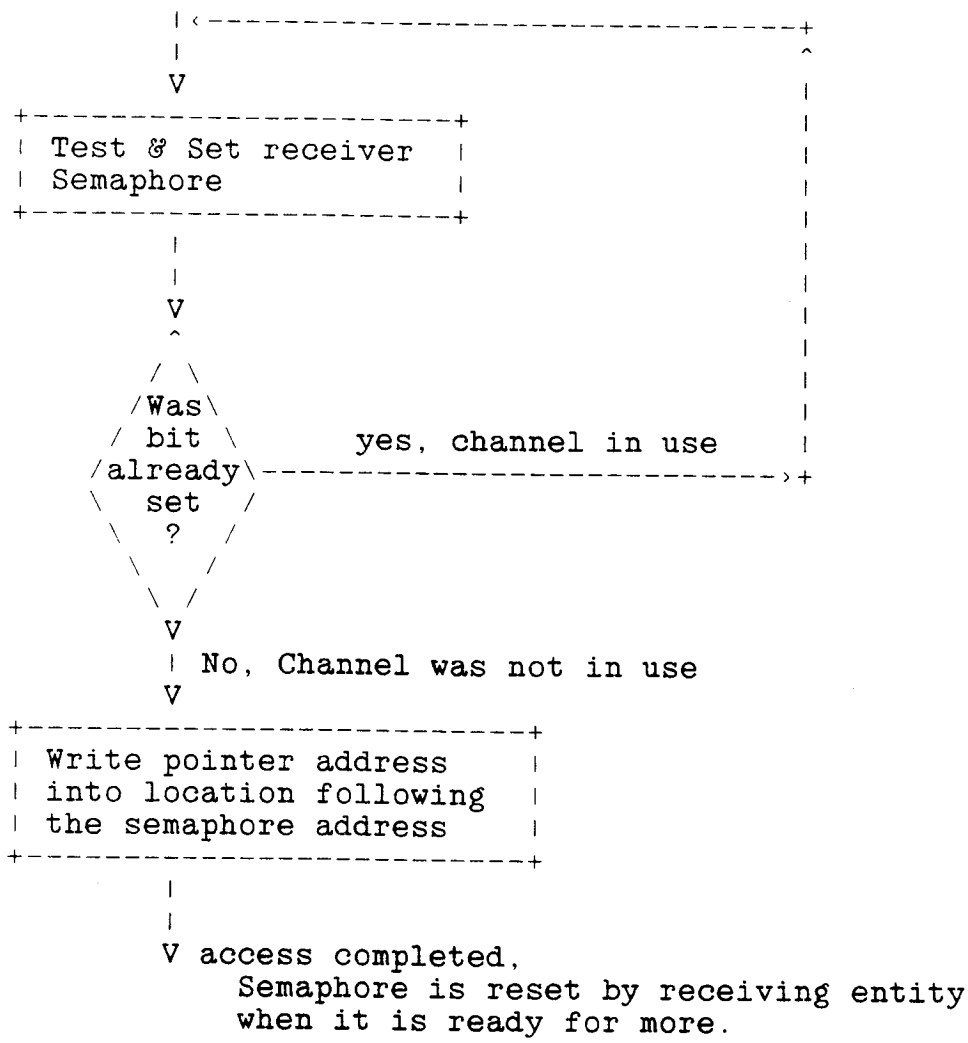


Figure 6.0

SETTING THE SEMAPHORE BIT

The VMEbus provides a read-modify-write capacity which should be used to implement the channel semaphore. The MII ICD requires the use of the semaphore in order to allow multiple users to access the same channel and avoid collisions. To give an example, if there are two parts of the LLC which have the ability to use a single MAC channel, then the owner of the channel must be established before it can be used. Otherwise both might attempt to use the channel at the same time. Any time one or more asynchronous devices wish to share a channel this decision must occur.

The winner of the right to own the channel is given to the one who first test and sets a bit within the confines of a single bus access. This bit is a semaphore bit and is called the channel busy bit. Each user tests the busy bit. If the test indicates that the channel is not busy then it sets the channel busy and uses it. The read-modify-write bus cycle is used because a problem may arise if two devices or entities test the busy bit at the same time before either has had the chance of setting it. Both would think that it was the owner of the channel. If the bit is tested and then set in a single non-interrupting bus access then no other device can test at the same time. Since it was tested and set in the same operation then a false indication says that it was previously not set and your the one who set it, therefore your the unique owner. The loser will test the bit as true, meaning the bit was previously set and it simply set it again. The Test-and-Set (68000) instruction has been used as an example because it typically causes the read-modify-write cycle on the VMEbus. It is not the only method which will insure proper semaphore testing.

RESETTING THE SEMAPHORE BIT

The MII channels are virtual interfaces and its actual implementations are transparent to the users. The detection and reset of the Semaphore bit is not specified by the MII and is left to the designer of the receiving entity. The MII allows the channel locations to be anywhere in the common memory map and therefore allows the designer to provide on board decoding of the semaphore location and its associated pointer. Such an implementation would appear to the MII bus as simple memory locations, but actually could be local memory and access might result in a local interrupt. The receiving channel could be located on a standard memory card and the semaphore bit polled by the receiving entity. To avoid the receiver and transmitter interleaving access to the channel, the address following the semaphore location could contain a pointer to itself. When the pointer is overwritten then the location can be considered valid.

The receiving entity is expected to reset the busy bit after it no longer needs the address passed to it.

The address passed to the receiver is a pointer to a record. This record will contain another address to another record like itself. When each record points to another record, it is referred to as a linked list. The last link in the list points to itself. Each record contains information written in a format as described in the Encoding section of this

document.

The channels are shown below:

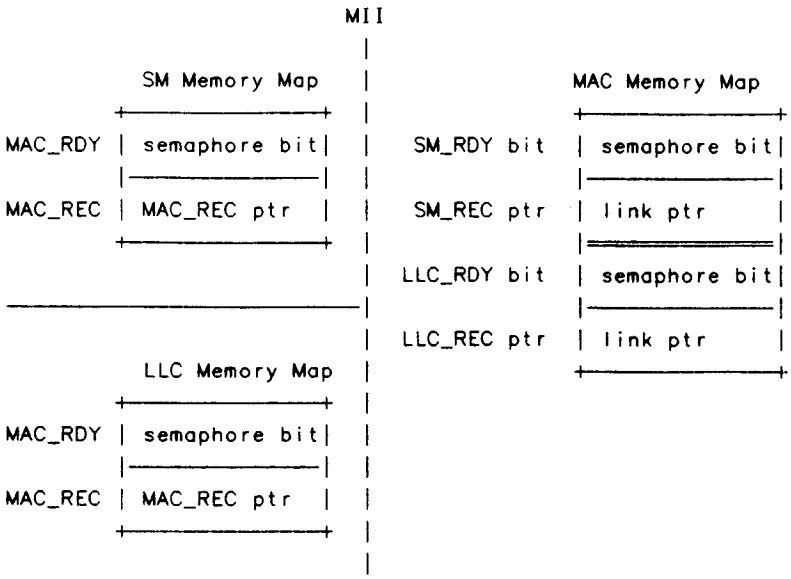


Figure 7.0 MII Memory Maps

MAC Memory Map Descriptions

SM_RDY bit - SM test this bit before writing into SM_REC
SM_REC ptr - SM writes pointer to link list of SM commands.

LLC_RDY bit - LLC test this bit before writing into LLC_REC
LLC_REC ptr _ LLC writes pointer to link list of LLC ICI primitives.

LLC Memory Map Descriptions

MAC_RDY bit - MAC test this bit before writing into MAC_REC in the LLC memory map.
MAC_REC ptr - MAC writes pointer to link list of MAC ICI primitives.

SM Memory Map Descriptions

MAC_RDY bit - MAC test this bit before writing into MAC_REC in the Station Manager memory map.
MAC_REC ptr - MAC writes pointer to link list of MAC ICI primitives.

3.8 SYNTAX

The common set of primitives to be used by MII are described with a syntax which is itself a standard. This standard is ISO 8824 ASN.1. The primitives and encodings are shown in the Primitive Syntax and Encoding section of this document. The primitives between the MAC and the LLC are rigidly defined and are common between the different MAC standards. The primitive descriptions were structured in such a way as to take advantage of useful options in the MACs (i.e. priorities, linking, etc) and to allow for future growth

3.9 ENCODING

The encoding used in the MII ICD can be found in ISO 8825, ASN.1 encoding. It is the ISO OSI standard encoding for the ASN.1 syntax. This encoding for ASN.1 has been adopted for the MII because it is an established and well known standard. As encoded, the MII primitives have been optimized for size and complexity. The encoding includes length bits which are only useful to the MII when there is confusion as to the quantity of information embedded in a single primitive.

3.10 CHANNEL RULES

A channel consist of two locations mapped to memory which is common to all entities on the MII. Both locations will be supported globally as read/write memory locations. The first location is the semaphore location and the second location is the Link location. Both address locations will respond to Long addressing as specified in the VMEbus specification. The Rules for channel operation are simple;

1) To use a channel; A non-interrupting test and set of the semaphore bit must be performed before each access to the location which follows it. This bit must test as not set before the set bus cycle occurs in order to claim access to it. In no other case can a write be made to the location following the semaphore.

2) The semaphore bit to be tested is the high order bit of a the lowest order byte as defined in the VMEbus specification. The bus access must be a read-modify-write bus cycle and is left to the implementer as to how the cycle is invoked (such as a Test & Set with a 68k series CPU).

3) A bus write of the location following the semaphore location will be supported as a normal VMEbus memory access (DTACK or BUSerror). Indiscriminate reads may not provide accurate answers. Once written to, ownership of the channel is no longer valid and the location may be overwritten by the receiving entity. Transmitting entities should consider it a write only location.

4) After a successful claim to the channel as outlined in item 1, a SINGLE write is allowed to the link location. The information written is to be an address where a MII record can be found. After writing it, item 1 must be repeated before the channel can be accessed again.

3.11 PRIMITIVES

3.11.1 MAC/LLC PRIMITIVES -

3.11.1.1 COMMANDS -

The LLC Interface supports MA_DATA request and confirms and MA_DATA indication and MA_DATA.indi_acks. Details of these commands are described in the IEEE 802.3 and IEEE 802.2 documents. A brief explanation follows;

```
MA_DATA.REQUEST
{ DESTINATION_ADDRESS, M_SDU, DESIRED_QUALITY }
```

This command comes from the LLC to the MAC and represents a request to ship the data pointed to by the M_SDU to the station at address DESTINATION_ADDRESS using a level of quality of DESIRED_QUALITY. The MAC is expected to respond with the following command;

```
MA_DATA.CONFIRMATION
{ QUALITY, STATUS }
```

This command from the MAC to the LLC will indicate to the LLC that the data previously requested to be shipped has been sent.

```
MA_DATA.INDICATION
{ DESTINATION_ADDRESS, SOURCE_ADDRESS,
  M_SDU, QUALITY                        }
```

This command from the MAC to the LLC will indicate to the LLC that data located at pointer M_SDU from the station SOURCE_ADDRESS was sent to DESTINATION_ADDRESS (needed to identify when a group address is used) with a QUALITY of service. The MAC expects the LLC to overwrite the MA_DATA.INDICATION with the MA_DATA.INDI_ACK thus allowing it to release the message buffer.

```
MA_DATA.INDI_ACK
{ STATUS }
```

This command from the LLC to the MAC will indicate to the MAC that the LLC has no more use for the indicate message buffer.

3.11.1.2 SYNTAX -

The LLC communicates to the MAC across the MII. The syntax is written according to Abstract Syntax Notation One or ASN.1 (ISO DIS 8824). The information described is encoded to the basic coding rules as found in ASN.1 (ISO DIS 8825).

```
Message_record ::= [PRIVATE 0] CHOICE
{ ma_data_request  [0]  Ma_request_type     |
  ma_data_confirm  [1]  Ma_confirm_type     |
  ma_data_indicate [2]  Ma_indicate-type    |
  ma_indi_ack      [3]  Ma_indi_ack_type    }

Ma_request_type ::= SET
{ destination_address [0] Net_address_type              ,
  M_SDU               [1] M_SDU_type                    ,
  requested_Ser_class [2] Req_ser_type                  ,
  frame_control       [3] Frame_con_type   (optional)   ,
  stream              [4] Stream_type      (optional)   ,
  link_list           [5] Link_list_type   (optional)   ,
  token_class         [6] Token_class_type (optional)
}

Ma_indicate-type ::= SET
{ destination_address [0] Net_address_type              ,
  source_address      [1] Net_address_type              ,
  M_SDU               [2] M_SDU_type                    ,
  reception_status    [3] Rec_status (optional)         ,
  requested_Ser_class [4] Req_ser_type (optional)       ,
  frame_control       [5] Frame_con_type   (optional)   ,
  link_list           [6] Link_list_type   (optional)
}
  -- The optional parameters have a default value.


Ma_confirm_type ::=  SET
{  transmit_status       [0]  Tran_status                    ,
   provided_ser_class    [1]  Provided_ser_type (optional),
   number_of_sdu_links   [2]  Number_of_sdu (optional),
   link_list             [3]  Link_list_type   (optional)
}


Ma_indi_ack_type ::=  SET
{ indi_status    [0] Integer, -- 1 = good 0= not accepted
  link_list      [2] Link_list_type   (optional)
 }

Net_address_type ::= CHOICE
{  net_add_16  Value_integer_1, -- 16 bit address option
   net_add_48  Value_integer_48 -- 48 bit address option
}

M_SDU_type ::= SET
{  SDU_PTR     [0]   Address,
   SDU_SIZE    [1]   INTEGER,
   buff_num    [2]   INTEGER
```

```
}

Req_ser_type ::= SET
{ priority        [0]   INTEGER, (optional)
  response        [1]   INTEGER, (optional) -- ack = 1
  quality_of_ser  [2]   INTEGER  (optional)
 }

Provided_ser_type ::= SET
{ priority        [0]   INTEGER, (optional)
  response        [1]   INTEGER, (optional) -- ack = 1
  quality_of_ser  [2]   INTEGER  (optional) }

Indi_ack_type ::= SET
{ indi_ack_status  Ack_status }

Frame_con_type ::= SET {} --TBD 2 octets, see FDDI 5.4.1

Link_list_type ::= SET {Address}

Stream_type ::= SET {INTEGER} -- 1= multiple M_SDUs xmitted
                                 See FDDI

Token_class_type ::= SET {INTEGER} --TBD

Rec_status ::= CHOICE
{  status [0] INTEGER -- 1 = good, all else bad.
 }

Tran_status ::= CHOICE
{  status [0] INTEGER -- 1 = good, all else bad.
}

Number_of_sdu ::= CHOICE {INTEGER} -- Number of M_SDUs
transmitted
```

3.11.2 STATION MANAGER -

3.11.2.1 COMMANDS -

The Station manager sends invoke commands to the MAC and the MAC responds with a reply response. The pairs which follow are first station manager command followed by the MAC response.

SM_MAC_LM_SET_VALUE.INVOKE
SM_MAC_LM_SET_VALUE.REPLY

SM_MAC_LM_GET_VALUE.INVOKE
SM_MAC_LM_GET_VALUE.REPLY

SM_MAC_LM_COMPARE_AND_SET_VALUE.INVOKE
SM_MAC_LM_COMPARE_AND_SET_VALUE.REPLY

SM_MAC_ACTION_VALUE.INVOKE
SM_MAC_ACTION_VALUE.REPLY

The Station manager can set an event mask which allows the MAC to report events without a direct request. The MAC can initiate a NOTIFY and expects a REPLY from the SM in response.

SM_MAC_EVENT_VALUE.NOTIFY
SM_MAC_EVENT_VALUE.REPLY

COMMAND DESCRIPTIONS

SM_MAC_LM_SET_VALUE.INVOKE
{ PARAMETER_TYPE, ACCESS_CONTROL_INFO }

The objective of the SM_MAC_LM_SET_VALUE.INVOKE command by the SM is to set a value in the MAC as defined by the parameter_type structure. This structure specifies both the variable to be set and the value to which it is set.

SM_MAC_LM_SET_VALUE.REPLY
{ STATUS }

The objective of the reply by the MAC to the SM is to indicate the success or failure of a previous SM_MAC_LM_SET_VALUE.INVOKE. The SM expects the MAC to overwrite the SM_MAC_LM_SET_VALUE.INVOKE with the SM_MAC_LM_SET_VALUE.REPLY thus allowing the SM to release the message buffer.

SM_MAC_LM_GET_VALUE.INVOKE
{ PARAMETER_TYPE, ACCESS_CONTROL_INFO }

The objective of the SM_MAC_LM_GET_VALUE.INVOKE command by the SM is to get a value in the MAC as defined by the parameter_type structure. This structure specifies the variable to be read.

SM_MAC_LM_GET_VALUE.REPLY
{ PARAMETER_TYPE, STATUS }

The objective of the reply by the MAC to the SM is to indicate the success or failure of a previous SM_MAC_LM_GET_VALUE.INVOKE. The SM expects the MAC to overwrite the SM_MAC_LM_GET_VALUE.INVOKE with the SM_MAC_LM_GET_VALUE.REPLY thus allowing the SM to release the message buffer.

SM_MAC_LM_COMPARE_AND_SET_VALUE.INVOKE
{ PARAMETER_TYPE,
OPERATION_COMMAND,
ACCESS_CONTROL_INFO }

The Compare and Set value command forces the MAC to do a comparison (of either a given constant or of a MAC variable) against a MAC variable. If the comparison is true then the MAC variable is over written. The PARAMETER_TYPE indicates the parameter to be over written and the value to use. The OPERATION_COMMAND structure specifies the comparison to do, and the constant or MAC variable to use in the comparison.

SM_MAC_LM_COMPARE_AND_SET_VALUE.REPLY
{ STATUS, RETURN_VAL }

The objective of the reply by the MAC to the SM is to indicate the success or failure of a previous SM_MAC_LM_COMPARE_AND_SET_VALUE.INVOKE. The SM expects the MAC to overwrite the SM_MAC_LM_COMPARE_AND_SET_VALUE.INVOKE with the SM_MAC_LM_COMPARE_AND_SET_VALUE.REPLY thus allowing the SM to release the message buffer.

SM_MAC_ACTION_VALUE.INVOKE
{ PARAMETER_ID, ACCESS_CONTROL_INFO }

The objective of the SM_MAC_ACTION_VALUE.INVOKE command by the SM is to force a MAC operation in the MAC as defined by the parameter_ID structure. This structure specifies the action to be performed.

SM_MAC_ACTION_VALUE.REPLY
{ STATUS, ACTION_REPORT };

The objective of the reply by the MAC to the SM is to indicate the success or failure of a previous SM_MAC_ACTION_VALUE.INVOKE . The SM expects the MAC to overwrite the SM_MAC_ACTION_VALUE.INVOKE with the SM_MAC_ACTION_VALUE.REPLY thus allowing the SM to release the message buffer.

MAC_SM_EVENT_VALUE.NOTIFY
{ EVENT_ID }

The objective of the MAC_SM_EVENT_VALUE.NOTIFY command by the MAC is to report a event which has occurred in the MAC as defined by the EVENT_ID structure. This structure specifies the Event and an integer. These events can be masked by setting the EVENT_MASK variable.

MAC_SM_EVENT_VALUE.REPLY
{ STATUS }

The objective of the following reply by the SM to the MAC is to indicate the success or failure of a previous MAC_SM_EVENT_VALUE.NOTIFY. The MAC expects the SM to overwrite the MAC_SM_EVENT_VALUE.NOTIFY with the MAC_SM_EVENT_VALUE.REPLY thus allowing the MAC to release the message buffer.

3.11.2.1.1 IEEE 802.3 SM MANAGEMENT -

See the IEEE 802 specifications for actual meanings. Some parameters have additional explanations. Specific implementations may have differences and the Station Manager must be able to resolve them. This set is taken from the IEEE 802 document and some implementations may not provide all the variables, however in such a case implementations will respond with a proper status (non-compliance or error, etc).

Additions may be made so as to support future changes providing that only new additions are made. Tags found in this standard may not be modified. New ones may be created and added as optional parameters.

```
READ_WRITE_VALUE_TYPES ::=  CHOICE
{  [0]       Mac_type                    |
   [1]       Memory                      |
   [2]       Slot_time                   |
   [3]       Inter_frame_gap             |
   [4]       Attempt_limit               |
   [5]       Back_off_limit              |
   [6]       Jam_size                    |
   [7]       Max_frame_size              |
   [8]       Min_frame_size              |
   [9]       Address_size                |
   [10]      Event_enable_mask           |
   [11]      Ma_group_address            |
   [12]      Ts
 }

Memory ::= SEQUENCE
{ ici_mem_link [0] Mem_block, -- list of free ICI blocks
  pdu_mem_link [1] Mem_block  -- list of free PDU blocks
}

Mem_block ::= SEQUENCE
{ block_size   INTEGER, -- size of each block
  block_ptr    Address  -- pointer to first word in block
}

Ts ::=  Value_address_1

This variable represents the address of this station.


Slot_time ::=  Value_integer_1

This variable represents the slot time of this station. This
is the maximum time this station must wait on another
station to respond to a transmission.

Event_enable_mask ::=  Event_enable_bits

Event_enable_bits ::= BIT STRING
{ low_ici_mem                     (0),
  low_pdu_mem                     (1),
  duplicate_address               (2),
  faulty_transmitter              (3),
  xmit_queue_threshold_exceeded   (4),
  receive_queue_threshold_exceeded (5),
  watch_dog_timeout               (6),
  max_retry_encountered           (7),
```

```
  bad_message_sent                (8),
                        -- Where 1 is enabled
 }
```

The MAC will report events when discovered and the appropriate bit is set in the MASK above. The event is reported only once whenever the actual occurrence is detected.

```
Attempt_limit ::=  Value_integer_1

Ma_group_address ::=  SEQUENCE
{ address_no      INTEGER,
  group_add       Value_address_1
}
```

The MAC can respond to a list of group addresses. This is the method for the Station Manager to tell the MAC which addresses are acceptable. The collection of valid group addresses can be thought of as an array where ADDRESS_NO is the index into the array and the GROUP_ADD is the actual address. This command will set this address as part of the group addresses (unless there all used up). Different implementations may limit the size of the array.

```
Mac_type ::= 03h
```

This variable is a read only variable and indicates which version of MAC is responding.

```
Inter_frame_gap ::= Value_integer_1

Back_off_limit ::= Value_integer_1

Jam_size ::= Value_integer_1

Max_frame_size ::= Value_integer_1

Min_frame_size ::= Value_integer_1

Address_size ::= Value_integer_1

Status_type ::=  CHOICE
{     undefined_error           [0]   Value_integer_1 |
      success                   [1]   Value_integer_1 |
      refuse_to_comply          [2]   Value_integer_1 |
      not_supported             [3]   Value_integer_1 |
      error_in_perfor           [4]   Value_integer_1 |
      not_available             [5]   Value_integer_1 |
      bad_parameter_id          [6]   Value_integer_1 |
      bad_parameter_operation   [7]   Value_integer_1 |
      bad_parameter_value       [8]   Value_integer_1 |
      bad_expected_value        [9]   Value_integer_1  }
```

These are responses to a command indicating the status of the command. Following are expected uses of these responses;

```
undefined_error - Request was not understood or no
                  appropriate error message available.
success - A successful operation has been completed.
refuse_to_comply - The operation was impossible or illegal.
not_supported - The operation is not supported or
                recognized.
error_in_perfor - A error was encountered
                  during operation.
not_available - Information is not yet
                available.
bad_parameter_id - Parameter ID was not
                   recognized.
bad_parameter_operation - Operation requested
                          was not recognized
bad_parameter_value - The Parameter
                      value was bad.
bad_expected_value  - The expected value was
                      illegal.

Event_types ::= IMPLICIT SEQUENCE
{ event_class          Event_class_types     }

     Event_class_types ::=  CHOICE
     { local      [0]    Event_identifier_types      |
       remote     [1]    Event_identifier_types       }
```

Events in this implementation are always LOCAL (as opposed to events that occurred in a remote node).

```
Event_identifier_types ::= CHOICE
{ low_ici_mem                       [0]  Value_address_1  |
  low_pdu_mem                       [1]  Value_address_1  |
  duplicate_address                 [2]  VALUE_INTEGER_1  |
  faulty_transmitter                [3]  VALUE_INTEGER_1  |
  xmit_queue_threshold_exceeded     [4]  VALUE_INTEGER_1  |
  receive_queue_threshold_exceeded  [5]  VALUE_INTEGER_1  |
  watch_dog_timeout                 [6]  VALUE_INTEGER_1  |
  max_retry_encountered             [7]  VALUE_INTEGER_1  |
  bad_message_sent                  [8]  Value_address_1  }
```

These events are reported upon the discovery of the following conditions;

low_ici_mem - Flagged when the MAC detects it has or is running out of ICI memory blocks.

low_pdu_mem - Flagged when the MAC detects it has or is running out of PCI memory blocks.

duplicate_address -
faulty_transmitter -
xmit_queue_threshold_exceeded -
receive_queue_threshold_exceeded - Flagged when the MAC cannot get buffer space for incoming data.

watch_dog_timeout - Flagged if the hardware watch dog timer expires.

max_retry_encountered - Flagged when a the max retry is encountered.

bad_message_sent -Flagged when the MAC discovers a message which does not agrees with its indicated structure size (i.e. bad length field).

```
Action_value_types ::= CHOICE
{ reset                [0]   Value_integer_1   }

OPERATION_COMMAND_TYPES ::= CHOICE
  {test_<<             [0]  READ_WRITE_VALUE_TYPES    |
   test_>>             [1]  Read_write_value_types    |
   test_==             [2]  Read_write_value_types    |
   test_<>             [3]  Read_write_value_types    |
   test_<=             [4]  Read_write_value_types    |
   test_>=             [5]  Read_write_value_types    |
   <<_given_constant   [6]  Given                     |
   >>_given_constant   [7]  Given                     |
   ==_given_constant   [8]  Given                     |
   <>_given_constant   [9]  Given                     |
   <=_given_constant   [10] Given                     |
   >=_given_constant   [11] Given                       }
```

The above operations expects a variable (we'll call var1) to be internal. The complete structure includes either a variable or constant which we'll call var2. The constant is used to overwrite Var1 in case the operation test true so in the case of two internal vars being tested a constant is

also passed in. The above operation commands imply the following:

```
test_<<  - if var1 << var2 then var1=constant
test_>>  - if var1 >> var2 then var1=constant
test_==  - if var1 == var2 then var1=constant
test_<>  - if var1 <> var2 then var1=constant
test_<=  - if var1 <= var2 then var1=constant
test_>=  - if var1 >= var2 then var1=constant
<<_given_constant - if var1 << constant then var1=constant
>>_given_constant - if var1 >> constant then var1=constant
==_given_constant - if var1 == constant then var1=constant
<>_given_constant - if var1 <> constant then var1=constant
<=_given_constant - if var1 <= constant then var1=constant
>=_given_constant - if var1 <= constant then var1=constant
```

Var1 is a MAC parameter to be tested (internal). Its value is always returned along with a status. Var2 is a MAC parameter (internal) or a constant (external) used in the comparison of Var1 (internal). Var1 always refers to a variable located in the MAC. Var2 is either located in the MAC (a compare of two internal variables) or as a constant (external) passed in. In all cases a true test forces Var1 to be a external constant.

```
Constant ::=  Value_integer_1

Value_integer_1 ::= IMPLICIT Long_word

Value_address_1 ::=  IMPLICIT Long_word (32 BITS)

Value_address_16 ::=  IMPLICIT ARRAY OF 16 Long_words
                                (32 BITS EACH)
```

3.11.2.1.1.1 SYNTAX - STATION MANAGER INTERFACE SYNTAX

The station manager communicates to the MAC across the MII. The syntax of such communication is described below according to Abstract Syntax Notation One or ASN.1 (ISO DIS 8824). The information described is encoded to the basic coding rules as found in ASN.1 (ISO DIS 8825). Some sample records follow the syntax notations.

3.11.2.1.1.2 -FORMAL SYNTAX SPECIFICATION -

```
Message_record ::= [PRIVATE 0] CHOICE
{ [0]  Sm_mac_lm_set_value.invoke                          |
  [1]  Sm_mac_lm_set_value.reply                           |
  [2]  Sm_mac_lm_get_value.invoke                          |
  [3]  Sm_mac_lm_get_value.reply                           |
  [4]  Sm_mac_lm_compare_and_set_value.invoke              |
  [5]  Sm_mac_lm_compare_and_set_value.reply               |
  [6]  Sm_mac_action_value.invoke                          |
  [7]  Sm_mac_action_value.reply                           |
  [8]  Sm_mac_event_value.notify                           |
  [9]  Sm_mac_event_value.reply                           }


Sm_mac_lm_set_value.invoke ::= IMPLICIT SEQUENCE
{ parameter_type        Read_write_value_types ,
  access_control_info   NULL                   }

Sm_mac_lm_set_value.reply ::= IMPLICIT SEQUENCE
{ Return_val          Read_write_value_types,
  status              Status_type}

Sm_mac_lm_get_value.invoke ::= IMPLICIT SEQUENCE
{ Parameter_type        Read_write_value_types ,
  access_control_info   NULL                   }

Sm_mac_lm_get_value.reply ::= IMPLICIT SEQUENCE
{ Parameter_type    Read_write_value_types ,
  status            Status_type }

Sm_mac_lm_compare_and_set_value.invoke ::= IMPLICIT SEQUENCE
{ parameter_type        Dummy_rw_types,
  operation_command     Operation_command_types,
  access_control_info   NULL                   }

Sm_mac_lm_compare_and_set_value.reply ::= IMPLICIT SEQUENCE
{  return_val           Read_write_value_types,
   status               Status_type }


Sm_mac_action_value.invoke ::= IMPLICIT SEQUENCE
{ parameter_id              Action_value_types ,
  access_control_info       NULL}

Sm_mac_action_value.reply ::= IMPLICIT SEQUENCE
{ status                Status_type,
  action_report         NULL    }
```

```
Mac_sm_event_value.notify ::= IMPLICIT SEQUENCE
{ Event_id              Event_types }

Mac_sm_event_value.reply ::= IMPLICIT SEQUENCE
{ Status                Status_type }


Read_write_value_types ::=  CHOICE
{  [0]      Mac_type                          |
   [1]      Memory                            |
   [2]      Slot_time                         |
   [3]      Inter_frame_gap                   |
   [4]      Attempt_limit                     |
   [5]      Back_off_limit                    |
   [6]      Jam_size                          |
   [7]      Max_frame_size                    |
   [8]      Min_frame_size                    |
   [9]      Address_size                      |
   [10]     Event_enable_mask                 |
   [11]     Ma_group_address                  |
   [12]     Ts

}

Dummy_rw_types ::=  CHOICE      {
mac_type                            [0] Value_integer_1  |
slot_time                           [2] Value_integer_1  |
inter_frame_gap                     [3] Value_integer_1  |
attempt_limit                       [4] Value_integer_1  |
back_off_limit                      [5] Value_integer_1  |
jam_size                            [6] Value_integer_1  |
max_frame_size                      [7] Value_integer_1  |
min_frame_size                      [8] Value_integer_1  |
address_size                        [9] Value_integer_1  |
event_enable_mask                   [10] Value_integer_1 |
ma_group_address                    [11] Value_integer_1 |
ts                                  [12] Value_address_1
}

Memory ::= SEQUENCE
{ ici_mem_link [0] Mem_block, -- list of free ICI blocks
  pdu_mem_link [1] Mem_block -- list of free PDU blocks
}

Mem_block ::= SEQUENCE
{ block_size   INTEGER, -- size of each block
  block_ptr    Address  -- pointer to first word in block
}
```

```
Ts ::=  Value_address_1

Ns ::=  Value_address_1

Slot_time ::=  Value_integer_1

Event_enable_mask ::=  EVENT_ENABLE_BITS

Ma_group_address ::=  SEQUENCE
{ Address_no      INTEGER,
  Group_add       Value_address_1
}

Mac_type ::= 03h

Status_type ::=  CHOICE
{     undefined_error           [0]   Value_integer_1 |
      success                   [1]   Value_integer_1 |
      refuse_to_comply          [2]   Value_integer_1 |
      not_supported             [3]   Value_integer_1 |
      error_in_prefor           [4]   Value_integer_1 |
      not_available             [5]   Value_integer_1 |
      bad_parameter_id          [6]   Value_integer_1 |
      bad_parameter_operation   [7]   Value_integer_1 |
      bad_parameter_value       [8]   Value_integer_1 |
      bad_expected_value        [9]   Value_integer_1  }

Event_types ::= IMPLICIT SEQUENCE
{ event_class          Event_class_types     }

Event_class_types ::=  CHOICE
{ local       [0]    Event_identifier_types          |
  remote      [1]    Event_identifier_types              }

Event_identifier_types ::= CHOICE
{ low_ici_mem                       [0]  Value_integer_1   |
  low_pdu_mem                       [1]  Value_integer_1   |
  duplicate_address                 [2]  Value_integer_1   |
  faulty_transmitter                [3]  Value_integer_1   |
  xmit_queue_threshold_exceeded     [4]  Value_integer_1   |
  receive_queue_threshold_exceeded  [5]  Value_integer_1   |
  watch_dog_timeout                 [6]  Value_integer_1   |
  max_retry_encountered             [7]  Value_integer_1   |
  bad_message_sent                  [8]  Value_address_1    }

Event_enable_bits ::= BIT STRING
{ low_ici_mem                       (0),
  low_pdu_mem                       (1),
  duplicate_address                 (2),
  faulty_transmitter                (3),
```

```
   xmit_queue_threshold_exceeded    (4),
   receive_queue_threshold_exceeded (5),
   watch_dog_timeout                (6),
   max_retry_encountered            (7),
   bad_message_sent                 (8) }— 1 is enabled

Action_value_types ::= CHOICE
{ reset              [0]   Value_integer_1          }

Operation_command_types ::= CHOICE
  {test_<<           [0]  Dummy_rw_types            |
   test_>>           [1]  Dummy_rw_types            |
   test_==           [2]  Dummy_rw_types            |
   test_<>           [3]  Dummy_rw_types            |
   test_<=           [4]  Dummy_rw_types            |
   test_>=           [5]  Dummy_rw_types            |
   <<_given_constant [6]  Given                     |
   >>_given_constant [7]  Given                     |
   ==_given_constant [8]  Given                     |
   <>_given_constant [9]  Given                     |
   <=_given_constant [10] Given                     |
   >=_given_constant [11] Given                      }

Given ::= CHOICE
  { [0] Value_integer_1       |
    [1] Value_address_1        }

Constant ::=  Value_integer_1

Value_integer_1 ::= IMPLICIT INTEGER

Value_address_1 ::=  IMPLICIT Long_word (32 BITS)

Value_address_16 ::=  IMPLICIT ARRAY OF 16 Long_words
                                (32 BITS EACH)
```

3.11.2.1.2 -IEEE 802.4 SM MANAGEMENT -

See the IEEE 802 specifications for actual meanings. Some parameters have additional explanations. Specific implementations may have differences and the Station Manager must be able to resolve them. This set is the is taken from the IEEE 802 document and some implementations may not provide a variable, however in such a case all implementations will respond with a status (non-compliance or error, etc).

Additions may be made so as to support future changes providing that only new additions are made. Tags found in this standard may not be modified. New ones may be created and added as optional parameters.

```
Read_write_value_types ::=  CHOICE        {
    [0]         Mac_type                                    |
    [1]         Memory                                      |
    [2]         Slot_time                                   |
    [3]         Hi_pri_token_hold_time                      |
    [4]         Max_ac_4_rotation_time                      |
    [5]         Max_ac_2_rotation_time                      |
    [6]         Max_ac_0_rotation_time                      |
    [7]         Mac_ring_maintenance_rotation_time          |
    [8]         Ring_maintenance_timer_initial_value        |
    [9]         Max_inter_solicit_count                     |
    [10]        Min_post_silence_preamble_length            |
    [11]        Event_enable_mask                           |
    [12]        Max_retry_limit                             |
    [13]        Ma_group_address                            |
    [15]        Channel_assignments                         |
    [16]        Transmitted_power_level_adjustment          |
    [17]        Transmitted_output_inhibits                 |
    [18]        Received_signal_sources                     |
    [19]        Signaling_mode                              |
    [20]        Received_signal_level_reporting             |
    [21]        Lan_topology_type                           |
    [22]        Ts                                          |
    [23]        Ns
}

Memory ::= SEQUENCE
{ ici_mem_link [0] Mem_block, -- list of free ICI blocks
  pdu_mem_link [1] Mem_block  -- list of free PDU blocks
}

Mem_block ::= SEQUENCE
{ block_size    INTEGER, -- size of each block
  block_ptr     Address  -- pointer to first word in block
}

Ts ::=  Value_address_1

This variable represents the address of this station.

Ns ::=  Value_address_1

This variable represents the address of the next station.

Slot_time ::=  Value_integer_1

Hi_pri_token_hold_time ::=  Value_integer_1

Max_ac_4_rotation_time ::=  Value_integer_1
```

```
Max_ac_2_rotation_time ::= Value_integer_1

Max_ac_0_rotation_time ::= Value_integer_1

Mac_ring_maintenance_rotation_time ::= Value_integer_1

Ring_maintenance_timer_initial_value ::= Value_integer_1

Max_inter_solicit_count ::= Value_integer_1

Min_post_silence_preamble_length ::= Value_integer_1

In_ring ::= Value_integer_1

Max_retry_limit ::= Value_integer_1

This is the maximum number of times that a packet will be
retransmitted when the acknowledgement indicates a bad
transmission. If this is a connection-less system this
variable should not be used.

Ma_group_address ::= SEQUENCE
{ address_no      INTEGER,
  group_add       Value_address_1 }

The MAC can respond to a list of group addresses. This is
the method for the Station Manager to tell the MAC which
addresses are acceptable. The collection of valid group
addresses can be thought of as an array where ADDRESS_NO is
the index into the array and the GROUP_ADD is the actual
address. This command will set this address as part of the
group addresses (unless there all used up). Different
implementations may limit the size of the array.

Channel_assignments ::= Value_integer_1

Transmitted_power_level_adjustment ::= Value_integer_1

Transmitted_output_inhibits ::= Value_integer_1

Received_signal_sources ::= Value_integer_1

Signaling_mode ::= Value_integer_1

Received_signal_level_reporting ::= Value_integer_1

Lan_topology_type ::= Value_integer_1

Mac_type ::= 04h
```

This variable is a read only variable and indicates which version of MAC is responding.

```
STATUS_TYPE ::=  CHOICE
{     undefined_error              [0]   Value_integer_1 |
      success                      [1]   Value_integer_1 |
      refuse_to_comply             [2]   Value_integer_1 |
      not_supported                [3]   Value_integer_1 |
      error_in_perfor              [4]   Value_integer_1 |
      not_available                [5]   Value_integer_1 |
      bad_parameter_id             [6]   Value_integer_1 |
      bad_parameter_operation      [7]   Value_integer_1 |
      bad_parameter_value          [8]   Value_integer_1 |
      bad_expected_value           [9]   Value_integer_1  }
```

These are responses to a command indicating the status of the command. Following are expected uses of these responses;

```
undefined_error - Request was not understood or no
                  appropriate error message available.
success - A successful operation has been completed.
refuse_to_comply - The operation was impossible or illegal.
not_supported - The operation is not supported or
                recognized.
error_in_perfor - A error was encountered
                  during operation.
not_available - Information is not yet
                available.
bad_parameter_id - Parameter ID was not
                   recognized.
bad_parameter_operation - Operation requested
                          was not recognized
bad_parameter_value - The Parameter
                      value was bad.
bad_expected_value  - The expected value was
                      illegal.
event_enable_mask ::=  EVENT_ENABLE_BITS

Event_enable_bits ::= BIT STRING
{ low_ici_mem                          (0),
  low_pdu_mem                          (1),
  duplicate_address                    (2),
  faulty_transmitter                   (3),
  xmit_queue_threshold_exceeded        (4),
  receive_queue_threshold_exceeded     (5),
  watch_dog_timeout                    (6),
  token_lost                           (8),
  dual_token                           (9),
  max_retry_encountered                (10),
  bad_message_sent                     (11),
```

```
  ns_changed      -                (12),
  ns_null                          (13), } — 1 is enabled
```

The MAC will report events when discovered and the appropriate bit is set in the MASK above. Bit 0 is the NS_Station, bit 1 is the NS_NULL etc. These bits are inspected each time the event has occurred and the MAC is active. The event is reported only once whenever the actual occurrence is detected.

```
Event_types ::= IMPLICIT SEQUENCE
{ event_class          Event_class_types     }

Event_class_types ::=  CHOICE
{ local       [0]    Event_identifier_types       |
  remote      [1]    Event_identifier_types         }
```

Events in the MII implementation are always LOCAL (as opposed
to events that occurred in a remote node).

```
Event_identifier_types ::= CHOICE
{ Low_ici_mem                       [0]  Value_integer_1   |
  Low_pdu_mem                       [1]  Value_integer_1   |
  Duplicate_address                 [2]  Value_integer_1   |
  Faulty_transmitter                [3]  Value_integer_1   |
  Xmit_queue_threshold_exceeded     [4]  Value_integer_1   |
  Receive_queue_threshold_exceeded  [5]  Value_integer_1   |
  Watch_dog_timeout                 [6]  Value_integer_1   |
  Token_lost                        [8]  Value_integer_1   |
  Dual_token                        [9]  Value_integer_1   |
  Max_retry_encountered             [10] Value_integer_1   |
  Bad_message_sent                  [11] Value_address_1   |
  Ns_null                           [12] Value_integer_1   |
  Ns_changed                        [13] Value_integer_1   }
```

These events are reported upon the discovery of the following conditions;

low_ici_mem - Flagged when the MAC detects it has or is running out of ICI memory blocks.

low_pdu_mem - Flagged when the MAC detects it has or is running out of PCI memory blocks.

ns_changed - Flagged when the event routine discovers a change in the NS address.

ns_null - Flagged when the NS is set to NULL

```
duplicate_address - reports duplicate addresses other
                    addresses.
faulty_transmitter - Reports faulty transmitter.

xmit_queue_threshold_exceeded - Flagged when the MAC
                                cannot get buffer space
                                for outgoing data.

receive_queue_threshold_exceeded - Flagged when the MAC
                                   cannot get buffer
                                   space for incoming
                                   data.

watch_dog_timeout - Flagged if the hardware watch dog
                    timer expires.

token_lost - Flagged when the token is detected as
             lost.

dual_token - Flagged when a extra token is discovered.

max_retry_encountered - Flagged when a the max retry is
                        encountered.

bad_message_sent -Flagged when the MAC discovers a message
                  which does not agrees with its indicated
                  structure size (i.e. bad length field).

Action_value_types ::= CHOICE
{ reset              [0]   Value_integer_1       }

The ACTION_VALUE_TYPES allow the following;

Reset Value_integer_1 = anything:

A reset will flush all queues, set all operating parameters
to their initial values, lose the token (if its holding it),
and await work from either the media or the LLC.

OPERATION_COMMAND_TYPES ::= CHOICE
  {test_<<            [0]  Read_write_value_types    |
   test_>>            [1]  Read_write_value_types    |
   test_==            [2]  Read_write_value_types    |
   test_<>            [3]  Read_write_value_types    |
   test_<=            [4]  Read_write_value_types    |
   test_>=            [5]  Read_write_value_types    |
   <<_given_constant  [6]  Given                     |
   >>_given_constant  [7]  Given                     |
   ==_given_constant  [8]  Given                     |
```

```
   <>_given_constant  [9]  Given                    |
   <=_given_constant  [10] Given                    |
   >=_given_constant  [11] Given                      }
```

The above operations expects a variable (we'll call var1) to be internal. The complete structure includes either a variable or constant which we'll call var2. The constant is used to overwrite Var1 in case the operation test true so in the case of two internal vars being tested a constant is also passed in. The above operation commands imply the following:

```
test_<<  - if var1 << var2 then var1=constant
test_>>  - if var1 >> var2 then var1=constant
test_==  - if var1 == var2 then var1=constant
test_<>  - if var1 <> var2 then var1=constant
test_<=  - if var1 <= var2 then var1=constant
test_>=  - if var1 >= var2 then var1=constant
<<_given_constant - if var1 << constant then var1=constant
>>_given_constant - if var1 >> constant then var1=constant
==_given_constant - if var1 == constant then var1=constant
<>_given_constant - if var1 <> constant then var1=constant
<=_given_constant - if var1 <= constant then var1=constant
>=_given_constant - if var1 <= constant then var1=constant
```

Var1 is a MAC parameter to be tested (internal). Its value is always returned along with a status. Var2 is a MAC parameter (internal) or a constant (external) used in the comparison of Var1 (internal). Var1 always refers to a variable located in the MAC. Var2 is either located in the MAC (a compare of two internal variables) or as a constant (external) passed in. In all cases a true test forces Var1 to be a external constant.

```
Constant ::=  Value_integer_1

Value_integer_1 ::= IMPLICIT Long_word

Value_address_1 ::=  IMPLICIT Long_word (32 BITS)

Value_address_16 ::=  IMPLICIT ARRAY OF 16 Long_words
                                (32 BITS EACH)
```

3.11.2.1.2.1 SYNTAX - STATION MANAGER INTERFACE SYNTAX

The station manager communicates to the MAC across the MII. The syntax of such communication is described below according to Abstract Syntax Notation One or ASN.1 (ISO DIS 8824). The information described is encoded to the basic coding rules as found in ASN.1 (ISO DIS 8825). Some sample records follow the syntax notations.

3.11.2.1.2.2 FORMAL SYNTAX SPECIFICATION -

```
Message_record ::= [PRIVATE 0] CHOICE
{ [0]  Sm_mac_lm_set_value.invoke                          |
  [1]  Sm_mac_lm_set_value.reply                           |
  [2]  Sm_mac_lm_get_value.invoke                          |
  [3]  Sm_mac_lm_get_value.reply                           |
  [4]  Sm_mac_lm_compare_and_set_value.invoke              |
  [5]  Sm_mac_lm_compare_and_set_value.reply               |
  [6]  Sm_mac_action_value.invoke                          |
  [7]  Sm_mac_action_value.reply                           |
  [8]  Sm_mac_event_value.notify                           |
  [9]  Sm_mac_event_value.reply                           }

Sm_mac_lm_set_value.invoke ::= IMPLICIT SEQUENCE
{ parameter_type        Read_write_value_types ,
  access_control_info   NULL                   }

Sm_mac_lm_set_value.reply ::= IMPLICIT SEQUENCE
{ return_val        Read_write_value_types,
  status                Status_type}

Sm_mac_lm_get_value.invoke ::= IMPLICIT SEQUENCE
{ parameter_type        Read_write_value_types ,
  access_control_info   NULL                   }

Sm_mac_lm_get_value.reply ::= IMPLICIT SEQUENCE
{ parameter_type    Read_write_value_types ,
  status            Status_type }

Sm_mac_lm_compare_and_set_value.invoke ::= IMPLICIT SEQUENCE
{ parameter_type        Dummy_rw_types,
  operation_command     Operation_command_types,
  access_control_info   NULL                   }

Sm_mac_lm_compare_and_set_value.reply ::= IMPLICIT SEQUENCE
{  return_val           Read_write_value_types,
   status               Status_type }

Sm_mac_action_value.invoke ::= IMPLICIT SEQUENCE
{ parameter_id              Action_value_types ,
  access_control_info       NULL}

Sm_mac_action_value.reply ::= IMPLICIT SEQUENCE
{ status                Status_type,
  action_report         NULL    }

Mac_sm_event_value.notify ::= IMPLICIT SEQUENCE
```

```
{ event_id          -        Event_types }

Mac_sm_event_value.reply ::= IMPLICIT SEQUENCE
{ status                     Status_type }

Read_write_value_types ::=  CHOICE        {
   [0]       Mac_type                                |
   [1]       Memory                                  |
   [2]       Slot_time                               |
   [3]       Hi_pri_token_hold_time                  |
   [4]       Max_ac_4_rotation_time                  |
   [5]       Max_ac_2_rotation_time                  |
   [6]       Max_ac_0_rotation_time                  |
   [7]       Mac_ring_maintenance_rotation_time      |
   [8]       Ring_maintenance_timer_initial_value    |
   [9]       Max_inter_solicit_count                 |
   [10]      Min_post_silence_preamble_length        |
   [11]      Event_enable_mask                       |
   [12]      Max_retry_limit                         |
   [13]      Ma_group_address                        |
   [15]      Channel_assignments                     |
   [16]      Transmitted_power_level_adjustment      |
   [17]      Transmitted_output_inhibits             |
   [18]      Received_signal_sources                 |
   [19]      Signaling_mode                          |
   [20]      Received_signal_level_reporting         |
   [21]      Lan_topology_type                       |
   [22]      Ts                                      |
   [23]      Ns                                       }

Memory ::= SEQUENCE
{ ici_mem_link [0] Mem_block, -- list of free ICI blocks
  pdu_mem_link [1] Mem_block  -- list of free PDU blocks }

Mem_block ::= SEQUENCE
{ block_size   INTEGER, -- size of each block
  block_ptr    Address  -- pointer to first word in block }

Dummy_rw_types ::=  CHOICE        {
mac_type                                  [0] Value_integer_1   |
ns                                        [1] Value_integer_1   |
slot_time                                 [2] Value_integer_1   |
hi_pri_token_hold_time                    [3] Value_integer_1   |
max_ac_4_rotation_time                    [4] Value_integer_1   |
max_ac_2_rotation_time                    [5] Value_integer_1   |
max_ac_0_rotation_time                    [6] Value_integer_1   |
mac_ring_maintenance_rotation_time        [7] Value_integer_1   |
ring_maintenance_timer_initial_value      [8] Value_integer_1   |
max_inter_solicit_count                   [9] Value_integer_1   |
min_post_silence_preamble_length          [10] Value_integer_1  |
```

```
event_enable_mask                    [11] Value_integer_1 |
max_retry_limit                      [12] Value_integer_1 |
ma_group_address                     [13] Value_integer_1 |
channel_assignments                  [15] Value_integer_1 |
transmitted_power_level_adjustment   [16] Value_integer_1 |
transmitted_output_inhibits          [17] Value_integer_1 |
received_signal_sources              [18] Value_integer_1 |
signaling_mode                       [19] Value_integer_1 |
received_signal_level_reporting      [20] Value_integer_1 |
lan_topology_type                    [21] Value_integer_1 |
ts                                   [22] Value_integer_1 }

Ts ::= Value_address_1

Ns ::= Value_address_1

Slot_time ::= Value_integer_1

Hi_pri_token_hold_time ::= Value_integer_1

Max_ac_4_rotation_time ::= Value_integer_1

Max_ac_2_rotation_time ::= Value_integer_1

Max_ac_0_rotation_time ::= Value_integer_1

Mac_ring_maintenance_rotation_time ::= Value_integer_1

Ring_maintenance_timer_initial_value ::= Value_integer_1

Max_inter_solicit_count ::= Value_integer_1

Min_post_silence_preamble_length ::= Value_integer_1

In_ring ::= Value_integer_1

Event_enable_mask ::= EVENT_ENABLE_BITS

Max_retry_limit ::= Value_integer_1

Ma_group_address ::= SEQUENCE
{ address_no      INTEGER,
  group_add       Value_address_1
}

Channel_assignments ::= Value_integer_1

Transmitted_power_level_adjustment ::= Value_integer_1

Transmitted_output_inhibits ::= Value_integer_1
```

```
Received_signal_sources ::= Value_integer_1

Signaling_mode ::= Value_integer_1

Received_signal_level_reporting ::= Value_integer_1

Lan_topology_type ::= Value_integer_1

Freeze_mac ::= Value_integer_1

Mac_type ::= 04h

Status_type ::= CHOICE
{   undefined_error           [0]  Value_integer_1 |
    success                   [1]  Value_integer_1 |
    refuse_to_comply          [2]  Value_integer_1 |
    not_supported             [3]  Value_integer_1 |
    error_in_prefor           [4]  Value_integer_1 |
    not_available             [5]  Value_integer_1 |
    bad_parameter_id          [6]  Value_integer_1 |
    bad_parameter_opereration [7]  Value_integer_1 |
    bad_parameter_value       [8]  Value_integer_1 |
    bad_expected_value        [9]  Value_integer_1 }

Event_types ::= IMPLICIT SEQUENCE
{ event_class          Event_class_types     }

EVENT_CLASS_TYPES ::= CHOICE
{ local     [0]   Event_identifier_types        |
  remote    [1]   Event_identifier_types            }

Event_identifier_types ::= CHOICE
{ low_ici_mem                       [0]  Value_integer_1  |
  low_pdu_mem                       [1]  Value_integer_1  |
  duplicate_address                 [2]  Value_integer_1  |
  faulty_transmitter                [3]  Value_integer_1  |
  xmit_queue_threshold_exceeded     [4]  Value_integer_1  |
  receive_queue_threshold_exceeded  [5]  Value_integer_1  |
  watch_dog_timeout                 [6]  Value_integer_1  |
  token_lost                        [8]  Value_integer_1  |
  dual_token                        [9]  Value_integer_1  |
  max_retry_encountered             [10] Value_integer_1  |
  bad_message_sent                  [11] Value_address_1  |
  ns_null                           [12] Value_integer_1  |
  ns_changed                        [13] Value_integer_1
}

Event_enable_bits ::= BIT STRING
{ low_ici_mem                       (0),
```

```
   low_pdu_mem                          (1),
   duplicate_address                    (2),
   faulty_transmitter                   (3),
   xmit_queue_threshold_exceeded        (4),
   receive_queue_threshold_exceeded     (5),
   watch_dog_timeout                    (6),
   token_lost                           (8),
   dual_token                           (9),
   max_retry_encountered                (10),
   bad_message_sent                     (11)
   ns_changed                           (12),
   ns_null                              (13)}   -- 1 is enabled

Action_value_types ::= CHOICE
{ reset                [0]   Value_integer_1    }

Operation_command_types ::= CHOICE
  {test_<<             [0]  Dummy_rw_types          |
   test_>>             [1]  Dummy_rw_types          |
   test_==             [2]  Dummy_rw_types          |
   test_<>             [3]  Dummy_rw_types          |
   test_<=             [4]  Dummy_rw_types          |
   test_>=             [5]  Dummy_rw_types          |
   <<_given_constant   [6]  Given                   |
   >>_given_constant   [7]  Given                   |
   ==_given_constant   [8]  Given                   |
   <>_given_constant   [9]  Given                   |
   <=_given_constant   [10] Given                   |
   >=_given_constant   [11] Given                    }


Given ::= CHOICE
  { [0] Value_integer_1       |
    [1] Value_address_1        }

Constant ::=  Value_integer_1

Value_integer_1 ::= IMPLICIT INTEGER

Value_address_1 ::=  IMPLICIT Long_word (32 BITS)

Value_address_16 ::=  IMPLICIT ARRAY OF 16 Long_words
                              (32 BITS EACH)
```

3.12 MII OPERATIONS

Although the actual process of initialization is not defined by the MII, it is outlined below.

1) The SM passes to the MAC (after checking the SM_RDY semaphore) a message which contains a pointer to a linked list of memory blocks suitable for the ICI information.

2) The SM passes to the MAC (after checking the SM_RDY semaphore) a message which contains a pointer to a linked list of memory blocks suitable for the PDU information.

3) The SM passes to the MAC (after checking the SM_RDY semaphore) a pointer to a linked list of memory blocks containing ASN.1 records which; a) tells the MAC the address of the SM receive channel (SM_REC) and semaphore (SM_RDY); b) request the MACs Status and type; c) and if appropriate puts it on line; all in a single linked list of commands. This also could be done with a series of single messages to the MAC.

LLC Initialization

The LLC initialization is beyond the MII scope except to say that the LLC must be made aware of the LLC_RDY semaphore and LLC_LINK locations.

OPERATIONS -Indicate

The SM operations with the MAC are no different than described under initialization. Each SM command has a unique link. Each SM command has a reply which overwrites the original command (the command record size is always larger than the reply). The reply indicates to the sender that it is now allowed to use that ICI memory block again. This way ICI can be passed back and forth without the need to request more blocks from the system. ICI information may include pointers to the PDUs and therefore a ICI reply also returns the PDU memory block to its original source.

When a PDU arrives from the media the MAC arbitrates for the bus and begins data movement to common memory using one of the free blocks of memory. The MAC design may or may not

completely buffer the data going into the common memory. If the MAC is the highest level priority on the VMEbus then a block mode operation will support the bandwidth necessary for no MAC buffering. These are design issues for the system designer. There are power, weight and speed advantages to no buffer MACs, however consuming the system bus for the length of one or more data packets may be unacceptable. The MII does not restrict the system in these areas.

Once the data is located in common memory the pointer to this PDU memory record and its size are included in a ASN.1 message known as MA_DATA.indicate. The MAC performs a TEST and SET operation on the LLCs MAC_RDY semaphore. If the Test indicates the bit was set, then the LLCs MAC channel was already busy and the set operation did nothing. In this case the MAC must wait. The MAC will continue to TEST and SET until the test indicates the busy bit was reset. The bit has already been set so the MAC is now allowed to use the channel. [The pointer to the ICI which is an encoded MA_DATA.indicate is written into the location following the semaphore. This indicates the presents of incoming data to the LLC (See IEEE 802.2).]

Once the MAC gains control over this channel it writes the pointer to the ICI (ASN.1 MA_DATA.indicate) record into the LLCs MAC_LINK location. When this location is written to, the LLC is interrupted. The LLC uses the address to find the ICI record and it points to the address of the PDU data located in common memory. The LLC will then queue the pointer, link the ICI record to an existing linked list (of previous ICI records) and frees the channel as soon as possible. LLC now holds the pointer to the ICI and PDU memory blocks.

The ICI memory (with the ASN.1 record in it) is then overwritten with a indicate acknowledge record (also ASN.1) and passed back to the MAC. The ICI also contains a pointer to the associated PDU. This allows the MAC to replenish its stock of free PDU blocks. The LLC and above layers must be finished with the PDU memory block before it sends the Indicate acknowledge primitive.

OPERATIONS -Request

A MA_DATA.request primitive is generated by the LLC to tell the MAC there is data to be shipped. This ASN.1 record is put into a ICI memory block, the LLC_RDY semaphore captured, and the pointer to the ICI block written to the LLC_LINK in the MAC. The MAC ships (or copies) the data, overwrites the

ICI block with a MA_DATA.confirm and passes it back to the LLC via the MAC_RDY semaphore and MAC_LINK locations in the LLC. The LLC can now replenish its stock of free ICI and PDU memory blocks.