FIRMWARE Manual
Eth29 / FDDI29 / MS360 / NICE-Eth

V1.7

N.A.T. GmbH

FIRMWARE Manual
Eth29 / FDDI29 / MS360 / NICE-Eth

V1.7

N.A.T. GmbH

1 Introduction

This document describes a software interface which allows any master cpu to communicate with one of N.A.T.s intelligent networking boards. The structure of these interface is allways the same for all boards. Normally this interface is represented by some memory locations within the dual port ram of the slave board. The access mechanism to these locations may differ from one board to another, but the overall handling is allways the same.

1.1 Structure of the Network Interface Field - "nif"

The host and the intelligent slaveboards Eth29/FDDI29 communicate with each other by a set of transaction fields and interrupts. The fields contain pointers to mbufs (small memory buffers) and are maintained on both sides within interrupt-routines. The fields are divided into two types: From Host to Slave (H2S) and from Slave to Host (S2H). Each field of the interface belongs to a queue which is maintained by the slaveboard or the host.
The fields are located at absolute memory addresses within the dual ported memory of the slaveboard.

mbuf : small memory buffer (128 Byte), used to transfer control information or data
page : large memory buffer, used to transfer data, possibly the final Ethernet/FDDI buffer (1536 Byte on Eth29, 4096 Byte on FDDI29)

Before taking an action the host must fill an mbuf taken from "mget", which then is returned to the slave by the ActionH2S-field. If the slave transfers an action to the host, the host must return the mbuf via "mput" after processing. The buffers are not transfered physicaly, only the addresses and their contents are transfered. When an action has been taken by one side, the field is cleared (or filled in the case of "mput", "pput") and the specific IRQ is generated.

1 Introduction

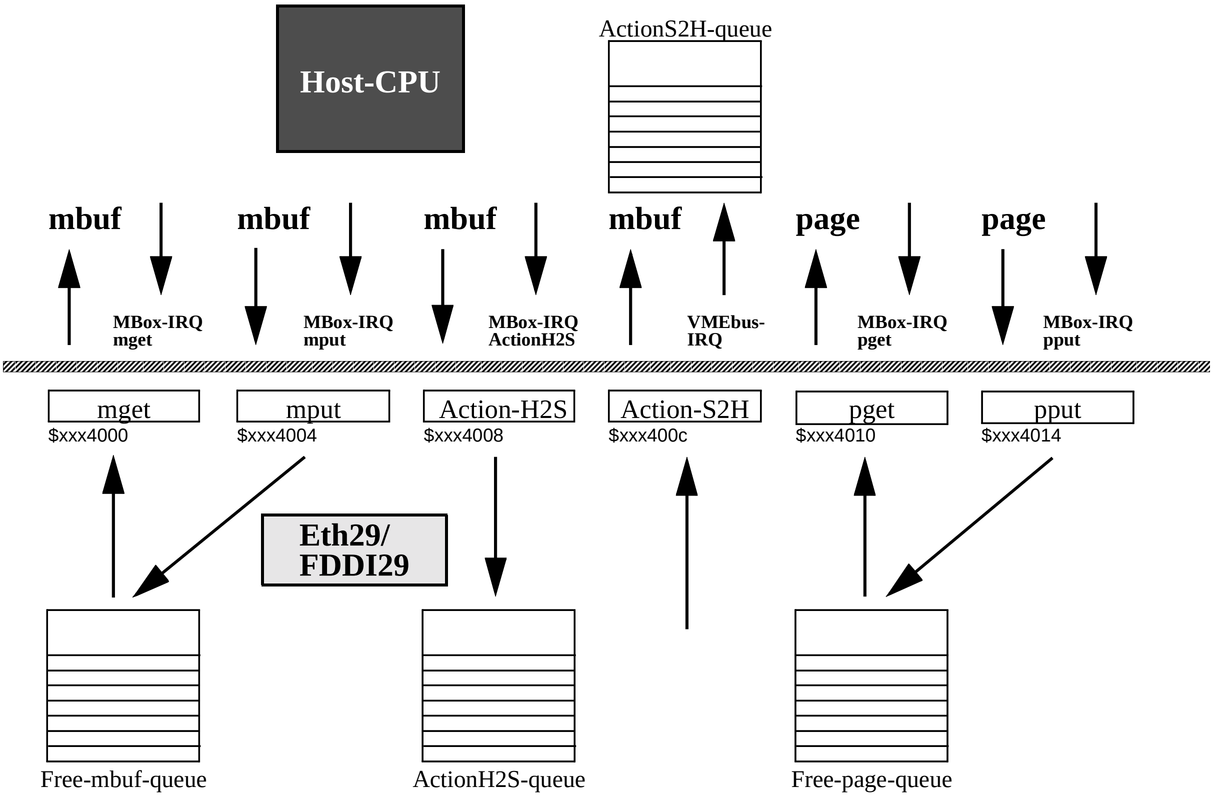
This document describes a software interface which allows any master cpu to communicate with one of N.A.T.s intelligent networking boards. The structure of these interface is allways the same for all boards. Normally this interface is represented by some memory locations within the dual port ram of the slave board. The access mechanism to these locations may differ from one board to another, but the overall handling is allways the same.

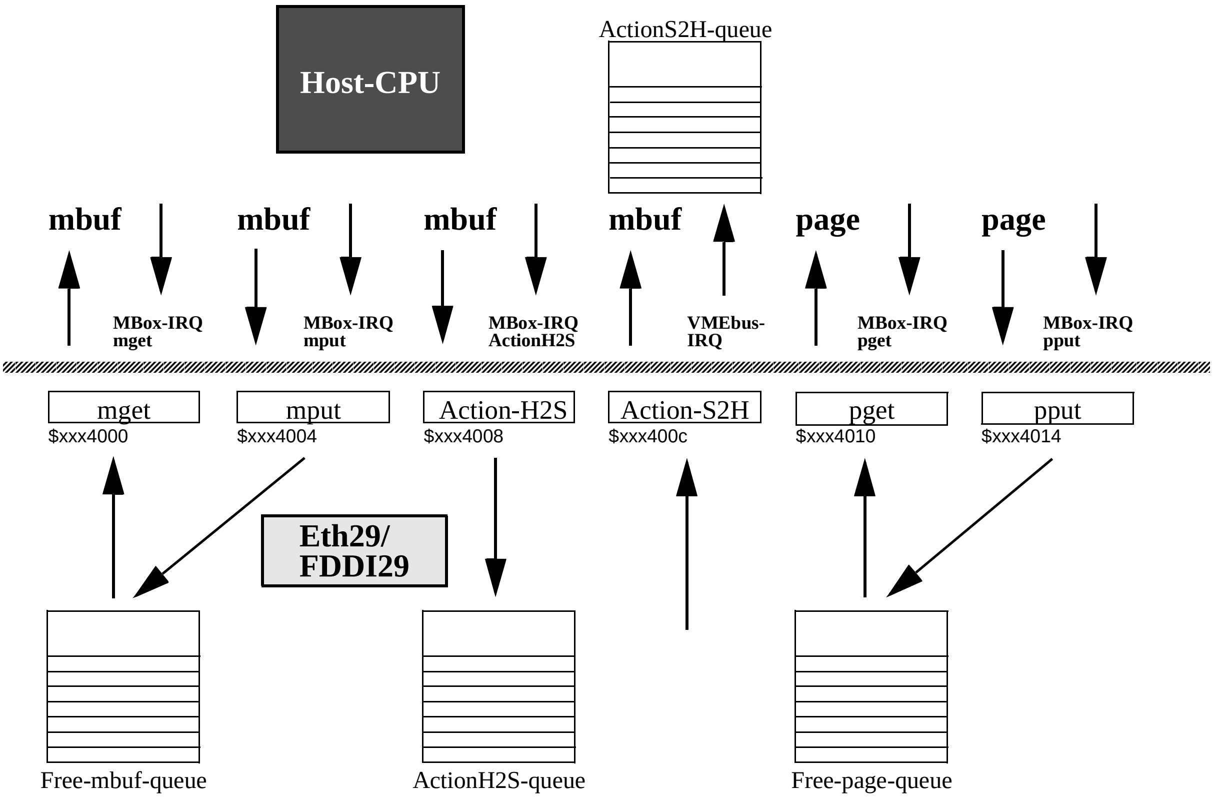
1.1 Structure of the Network Interface Field - "nif"

The host and the intelligent slaveboards Eth29/FDDI29 communicate with each other by a set of transaction fields and interrupts. The fields contain pointers to mbufs (small memory buffers) and are maintained on both sides within interrupt-routines. The fields are divided into two types: From Host to Slave (H2S) and from Slave to Host (S2H). Each field of the interface belongs to a queue which is maintained by the slaveboard or the host.
The fields are located at absolute memory addresses within the dual ported memory of the slaveboard.

mbuf : small memory buffer (128 Byte), used to transfer control information or data
page : large memory buffer, used to transfer data, possibly the final Ethernet/FDDI buffer (1536 Byte on Eth29, 4096 Byte on FDDI29)

Before taking an action the host must fill an mbuf taken from "mget", which then is returned to the slave by the ActionH2S-field. If the slave transfers an action to the host, the host must return the mbuf via "mput" after processing. The buffers are not transfered physicaly, only the addresses and their contents are transfered. When an action has been taken by one side, the field is cleared (or filled in the case of "mput", "pput") and the specific IRQ is generated.


Host-CPU
ActionS2H-queue
mbuf
mbuf
mbuf
mbuf
page
page
MBox-IRQ mget
MBox-IRQ mput
MBox-IRQ ActionH2S
VMEbus-IRQ
MBox-IRQ pget
MBox-IRQ pput
mget
$xxx4000
mput
$xxx4004
Action-H2S
$xxx4008
Action-S2H
$xxx400c
pget
$xxx4010
pput
$xxx4014
Eth29/
FDDI29
Free-mbuf-queue
ActionH2S-queue
Free-page-queue


Host-CPU
ActionS2H-queue
mbuf
mbuf
mbuf
mbuf
page
page
MBox-IRQ mget
MBox-IRQ mput
MBox-IRQ ActionH2S
VMEbus-IRQ
MBox-IRQ pget
MBox-IRQ pput
mget
$xxx4000
mput
$xxx4004
Action-H2S
$xxx4008
Action-S2H
$xxx400c
pget
$xxx4010
pput
$xxx4014
Eth29/
FDDI29
Free-mbuf-queue
ActionH2S-queue
Free-page-queue

2 Definition of the network interface (NIF) structure

A VMEBus masterboard and an intelligent slaveboard communicate through a data structure, which is situated in the multiported RAM area of the slaveboard. This is valid for all N.A.T. network protocols.
The network interface has following structure:

```
typedef struct nif_struct {
   struct mbuf     *mget;
   struct mbuf     *mput;
   struct mbuf     *ActH2S;
   struct mbuf     *ActS2H;
   unsigned char   *pget;
   unsigned char   *pput;
} NIF;
```

The fields of the **NIF**-structure have the following detailed meaning:

mget Hands over the adress of a mbuf on the slaveboard requested by the masterboard. Usually there is always available a free mbuf. If the free mbuf is used, the field mget has to be cleared and a mailbox interrupt has to be generated to notify the slaveboard, that a new mbuf has to be entered. (see function **put_mbuf()** in nif_a.a).

mput Hands over the address of a mbuf, which can be released on the slaveboard. After the address has been written into mput, the corresponding mailbox interrupt on the slaveboard has to be generated. The field will be cleared by the slaveboard (see function **put_mbuf()** in nif_a.a).

ActH2S (Action-Host-to-Slave)
Hands over of the address of a mbuf, which containes the parameter for carrying out a command on the slaveboard. The used mbuf had to be requested by mget first. The mbuf is released by the slaveboard. For the handling of the parameters the nifpar-structure described below is being used (see function **action()** in nif_a.a).

ActS2H (Action-Slave-to-Host)
Hands over the address of a mbuf from the slaveboard to the masterboard. This action is initiated by the slaveboard generating an VMEbus-interrupt. The mbuf can either contain the result of a command from the masterboard, or it contains the parameter of a command from the slaveboard to the masterboard. In both cases the slaveboard will cause a VMEbus interrupt. The interrupt routine on the Masterboard is responsible for the field **ACTS2H** becoming free as soon as possible, so that it can be used again by the slaveboard. The masterboard is responsible for the release of the mbuf. Again the containance of the nifpar-structure is used

2 Definition of the network interface (NIF) structure

A VMEBus masterboard and an intelligent slaveboard communicate through a data structure, which is situated in the multiported RAM area of the slaveboard. This is valid for all N.A.T. network protocols.
The network interface has following structure:

```
typedef struct nif_struct {
   struct mbuf      *mget;
   struct mbuf      *mput;
   struct mbuf      *ActH2S;
   struct mbuf      *ActS2H;
   unsigned char    *pget;
   unsigned char    *pput;
} NIF;
```

The fields of the **NIF**-structure have the following detailed meaning:

mget Hands over the adress of a mbuf on the slaveboard requested by the masterboard. Usually there is always available a free mbuf. If the free mbuf is used, the field mget has to be cleared and a mailbox interrupt has to be generated to notify the slaveboard, that a new mbuf has to be entered. (see function **put_mbuf()** in nif_a.a).

mput Hands over the address of a mbuf, which can be released on the slaveboard. After the address has been written into mput, the corresponding mailbox interrupt on the slaveboard has to be generated. The field will be cleared by the slaveboard (see function **put_mbuf()** in nif_a.a).

ActH2S (Action-Host-to-Slave)
Hands over of the address of a mbuf, which containes the parameter for carrying out a command on the slaveboard. The used mbuf had to be requested by mget first. The mbuf is released by the slaveboard. For the handling of the parameters the nifpar-structure described below is being used (see function **action()** in nif_a.a).

ActS2H (Action-Slave-to-Host)
Hands over the address of a mbuf from the slaveboard to the masterboard. This action is initiated by the slaveboard generating an VMEbus-interrupt. The mbuf can either contain the result of a command from the masterboard, or it contains the parameter of a command from the slaveboard to the masterboard. In both cases the slaveboard will cause a VMEbus interrupt. The interrupt routine on the Masterboard is responsible for the field **ACTS2H** becoming free as soon as possible, so that it can be used again by the slaveboard. The masterboard is responsible for the release of the mbuf. Again the containance of the nifpar-structure is used

(see example interrupt routine in isockdrv.a).

pget Hands over of the address of a page requested from the slaveboard. Usually there is always available a free page. When this free page is used, the field **pget** has to be cleared and a mailbox interrupt has to be generated. The address of a new page entered by the slaveboard (see function **get_page()** in nif_a.a).

pput Hands over the address of a page, which is returned back to the slaveboard. After the address has been written to pput, the corresponding mailbox interrupt has to be generated. The field will be cleared by the slaveboard. (see function **put_page()** in nif_a.a).

For each of the described fields is available a special interrupt code, which has to be written into the mailbox cell of the slaveboard to cause the treatment of the corresponding field by the slaveboard (s. MBox-IRQ - Codes).
For the exchange of controll information and datas between the masterboard and the slaveboard are available two basic buffer elements:

mbuf A small memory area with a length of 128 Bytes, which mainly hands over control-information, but which can be used for normal data-transfer as well.

page A large linear memory area for the transmission of the actual datas (for example network packages). This area has the size of 1536 Bytes for Ethernet packages and 4096 Bytes for FDDI packages.

All these memory elements and structures are physically situated in the multiported RAM area of the slaveboard.

The **mbuf**-structure used in the **NIF**-field has following form:

```
struct mbuf {
   struct mbuf     *m_next;              /* next buffer in chain */
   unsigned long   m_off;                /* offset of data */
   short           m_len;                /* amount of data in */
                                         /* this mbuf */
   short           m_type;               /* mbuf type (0 == free) */
   unsigned char   m_dat[MLEN];          /* data storage */
   struct mbuf     *m_act;               /* link in higher-level */
                                         /* mbuf list */
};
```

The field **m_off** may point either to the data area contained within the mbuf or to external datas contained within a page.

After the performance of following macros, the data-area within the mbuf can be accessed.

```
#define  mtod(x,t)    ((t)((int)(x) + (x)->m_off))
```

Example:

(see example interrupt routine in isockdrv.a).

pget Hands over of the address of a page requested from the slaveboard. Usually there is always available a free page. When this free page is used, the field **pget** has to be cleared and a mailbox interrupt has to be generated. The address of a new page entered by the slaveboard (see function **get_page()** in nif_a.a).

pput Hands over the address of a page, which is returned back to the slaveboard. After the address has been written to pput, the corresponding mailbox interrupt has to be generated. The field will be cleared by the slaveboard. (see function **put_page()** in nif_a.a).

For each of the described fields is available a special interrupt code, which has to be written into the mailbox cell of the slaveboard to cause the treatment of the corresponding field by the slaveboard (s. MBox-IRQ - Codes).
For the exchange of controll information and datas between the masterboard and the slaveboard are available two basic buffer elements:

mbuf A small memory area with a length of 128 Bytes, which mainly hands over control-information, but which can be used for normal data-transfer as well.

page A large linear memory area for the transmission of the actual datas (for example network packages). This area has the size of 1536 Bytes for Ethernet packages and 4096 Bytes for FDDI packages.

All these memory elements and structures are physically situated in the multiported RAM area of the slaveboard.

The **mbuf**-structure used in the **NIF**-field has following form:

```
struct mbuf {
   struct mbuf     *m_next;              /* next buffer in chain */
   unsigned long   m_off;                /* offset of data */
   short           m_len;                /* amount of data in */
                                         /* this mbuf */
   short           m_type;               /* mbuf type (0 == free) */
   unsigned char   m_dat[MLEN];          /* data storage */
   struct mbuf     *m_act;               /* link in higher-level */
                                         /* mbuf list */
};
```

The field **m_off** may point either to the data area contained within the mbuf or to external datas contained within a page.

After the performance of following macros, the data-area within the mbuf can be accessed.

```
#define  mtod(x,t)    ((t)((int)(x) + (x)->m_off))
```

Example:

```
struct nifpar   *nifp;
struct mbuf     *m;

nifp = mtod(m, struct nifpar *);
```

```
struct nifpar   *nifp;
struct mbuf     *m;

nifp = mtod(m, struct nifpar *);
```

3 The nifpar structure

The nifpar structure is the structure which is used together with the fields **ActH2S** respectively **ActS2H** (see above) for the commands from the masterboard to the slaveboard and for the corresponding answers from the slaveboard or for commands from the slaveboard to the masterboard. Because of the structures which are used here and which are subject to further developments or adjustments to new requirements, the file "nif.h", which defines all the used structures and constant factors, should be consulted as a programming reference.

```
struct nifpar {
   unsigned long     Command;
   unsigned long     Status;
   unsigned long     Errno;
   unsigned short    PID;
   unsigned short    pad;                      /* alignment on 4 byte */
                                               /* boundary */
   union {
           struct L2M_init_p          init;
           struct L2C_attp_p          attp;
           struct L2C_addt            addt;
           struct L2C_addvect         addvect;
           struct L2_send             L2_send;
           struct L2_rcv              L2_rcv;
           struct L2_attproto         L2_attproto;
           struct BCB                 bcb;
           struct S_CopyData          cpdat;
           struct S_Selwakeup         S_Selwakeup;
           struct Par_Socket          Par_Socket;
           struct Par_Bind            Par_Bind;
           struct Par_Listen          Par_Listen;
           struct Par_Accept          Par_Accept;
           struct Par_Connect         Par_Connect;
           struct Par_Sendto          Par_Sendto;
           struct Par_Send            Par_Send;
           struct Par_Sendmsg         Par_Sendmsg;
           struct Par_Recvfrom        Par_Recvfrom;
           struct Par_Recv            Par_Recv;
           struct Par_Recvmsg         Par_Recvmsg;
           struct Par_Shutdown        Par_Shutdown;
           struct Par_Setsockopt      Par_Setsockopt;
           struct Par_Getsockopt      Par_Getsockopt;
           struct Par_Getsockname     Par_Getsockname;
           struct Par_Getpeername     Par_Getpeername;
           struct Par_Ioctl           Par_Ioctl;
           struct Par_Close           Par_Close;
           struct Par_Select          Par_Select;
           struct Par_Getstataddr     Par_Getstataddr;
           unsigned char              ethid[6];
           unsigned long              debuglevel;
           struct meminfo             meminfo;
           struct taskinfo            taskinfo[12];
           unsigned long              license;
           struct sysreqmem           sysreqmem;
           struct sysretmem           sysretmem;
           struct Par_Sendx           Par_Sendx;
           struct ParCamCmd           Par_CamCmd;
   } Opt;
```

3 The nifpar structure

The nifpar structure is the structure which is used together with the fields **ActH2S** respectively **ActS2H** (see above) for the commands from the masterboard to the slaveboard and for the corresponding answers from the slaveboard or for commands from the slaveboard to the masterboard. Because of the structures which are used here and which are subject to further developments or adjustments to new requirements, the file "nif.h", which defines all the used structures and constant factors, should be consulted as a programming reference.

```
struct nifpar {
   unsigned long     Command;
   unsigned long     Status;
   unsigned long     Errno;
   unsigned short    PID;
   unsigned short    pad;                     /* alignment on 4 byte */
                                              /* boundary */
   union {
            struct L2M_init_p          init;
            struct L2C_attp_p          attp;
            struct L2C_addt            addt;
            struct L2C_addvect         addvect;
            struct L2_send             L2_send;
            struct L2_rcv              L2_rcv;
            struct L2_attproto         L2_attproto;
            struct BCB                 bcb;
            struct S_CopyData          cpdat;
            struct S_Selwakeup         S_Selwakeup;
            struct Par_Socket          Par_Socket;
            struct Par_Bind            Par_Bind;
            struct Par_Listen          Par_Listen;
            struct Par_Accept          Par_Accept;
            struct Par_Connect         Par_Connect;
            struct Par_Sendto          Par_Sendto;
            struct Par_Send            Par_Send;
            struct Par_Sendmsg         Par_Sendmsg;
            struct Par_Recvfrom        Par_Recvfrom;
            struct Par_Recv            Par_Recv;
            struct Par_Recvmsg         Par_Recvmsg;
            struct Par_Shutdown        Par_Shutdown;
            struct Par_Setsockopt      Par_Setsockopt;
            struct Par_Getsockopt      Par_Getsockopt;
            struct Par_Getsockname     Par_Getsockname;
            struct Par_Getpeername     Par_Getpeername;
            struct Par_Ioctl           Par_Ioctl;
            struct Par_Close           Par_Close;
            struct Par_Select          Par_Select;
            struct Par_Getstataddr     Par_Getstataddr;
            unsigned char              ethid[6];
            unsigned long              debuglevel;
            struct meminfo             meminfo;
            struct taskinfo            taskinfo[12];
            unsigned long              license;
            struct sysreqmem           sysreqmem;
            struct sysretmem           sysretmem;
            struct Par_Sendx           Par_Sendx;
            struct ParCamCmd           Par_CamCmd;
   } Opt;
```

```
};
```

The fields of the nifpar structure have the following detailed information:

Command A command from the masterboard to the slaveboard or from the slaveboard to the masterboard. The possible commands are described further below. The command numbers 1 to 100 are intended for the commands from the masterboard to the slaveboard, the numbers from 101 to 200 are intended for commands from the slaveboard to the masterboard.
The command word has the following sub structure:

Bits 24 - 31	Bits 16 - 23	Bits 8 - 15	Bits 0 - 7
0	***0***	***Port-ID***	***COM-ID***

The **COM-ID** is the command number and the **Port-ID** is the port identifier for the protocol issuing the command. All layer 2 commands must use **Port-ID** = 0.

Status Contains the return value of the performed function after the performance of a command from the masterboard to the slaveboard.

Errno When a command from the masterboard to the slaveboard causes an error, here is written the corresponding error number.

PID Has to be filled with the ID of the process, which starts the command when a command from the masterboard to the slaveboard is called up. When the slaveboard sends back the answer for this command, the process ID is unchanged written in this structure. Thus the process can be informed by the interrupt routine on the masterboard that the command has been completed. while the masterboard is treating the interrupt caused by the slaveboard.

pad Fillword, which serves for the alignment of an address divisible by 4.

Opt A Union, which contains a suitable data structure for each command. The description of the structures can be found further up.

```
};
```

The fields of the nifpar structure have the following detailed information:

Command A command from the masterboard to the slaveboard or from the slaveboard to the masterboard. The possible commands are described further below. The command numbers 1 to 100 are intended for the commands from the masterboard to the slaveboard, the numbers from 101 to 200 are intended for commands from the slaveboard to the masterboard.
The command word has the following sub structure:

Bits 24 - 31	Bits 16 - 23	Bits 8 - 15	Bits 0 - 7
0	***0***	***Port-ID***	***COM-ID***

The **COM-ID** is the command number and the **Port-ID** is the port identifier for the protocol issuing the command. All layer 2 commands must use **Port-ID** = 0.

Status Contains the return value of the performed function after the performance of a command from the masterboard to the slaveboard.

Errno When a command from the masterboard to the slaveboard causes an error, here is written the corresponding error number.

PID Has to be filled with the ID of the process, which starts the command when a command from the masterboard to the slaveboard is called up. When the slaveboard sends back the answer for this command, the process ID is unchanged written in this structure. Thus the process can be informed by the interrupt routine on the masterboard that the command has been completed. while the masterboard is treating the interrupt caused by the slaveboard.

pad Fillword, which serves for the alignment of an address divisible by 4.

Opt A Union, which contains a suitable data structure for each command. The description of the structures can be found further up.

3.1 Commands from the masterboard to the slaveboard

By the help of the cell **ActH2S** commands from the masterboard to the slaveboard can be transmitted. To each command belongs a structure, which will be written in the mbuf, whose address is handed over to the slaveboard with **ActH2S**.
After a mbuf has been requested with the function **get_mbuf()**, the data part is filled with the **nifpar** structure, whereby the structure corresponding to the commnd has to be used within the **Opt** union of the **nifpar** structure. The used mbuf is released by the slaveboard for one way commands. For commands expecting an answer or result, the same mbuf is used to handle over the return values in an **ActsS2H** command.
The fields **Status** and **Errno** of the **nifpar** structure are now filled with the corresponding values. The field **PID** remains unchanged.

3.1.1 LAYER 2 - Action Commands - Controll Commands

L2_INIT

The command **L2_INIT** is used for the initialisation of the slaveboard. The command is directly passed onto Layer 2. It uses following structure:

```
struct L2M_init_p {
   unsigned short  L_mode;          /* Lance operation mode */
   unsigned char   my_addr[6];      /* The own ETHERNET ID */
                                    /* overrides the ONBOARD */
                                    /* Ethernet ID */
   unsigned char   **multi_addr;    /* pointer to a list of */
                                    /* Multicast addresses */
   unsigned long   License;         /* the License number */
                                    /* of the board  */
   unsigned short  dma_modes;       /* DMA mode */
   unsigned long   a32_base;        /* base address of master board */
   unsigned long   a64_base;        /* high lword of master base if A64 */
};
```

Usage of L_mode with FDDI29:
Bridging Support:
For usage of the FDDI29 in bridging applications some configuration bits of the FDDI controller chipset have been made available at the users interface. Setting of these bits should be handled with great care. For detailed information of the meaning of these bits one should refer to the NS FDDI controller manuals.

- 0x0010 - set BOSEL bit of MAC MCMR2 register
- 0x0020 - set VST bit of BSI R1CR0 register (tx channel 1)
- 0x0040 - set SAT bit of BSI R1CR0 register (tx channel 1)

3.1 Commands from the masterboard to the slaveboard

By the help of the cell **ActH2S** commands from the masterboard to the slaveboard can be transmitted. To each command belongs a structure, which will be written in the mbuf, whose address is handed over to the slaveboard with **ActH2S**.
After a mbuf has been requested with the function **get_mbuf()**, the data part is filled with the **nifpar** structure, whereby the structure corresponding to the commnd has to be used within the **Opt** union of the **nifpar** structure. The used mbuf is released by the slaveboard for one way commands. For commands expecting an answer or result, the same mbuf is used to handle over the return values in an **ActsS2H** command.
The fields **Status** and **Errno** of the **nifpar** structure are now filled with the corresponding values. The field **PID** remains unchanged.

3.1.1 LAYER 2 - Action Commands - Controll Commands

L2_INIT

The command **L2_INIT** is used for the initialisation of the slaveboard. The command is directly passed onto Layer 2. It uses following structure:

```
struct L2M_init_p {
   unsigned short  L_mode;         /* Lance operation mode */
   unsigned char   my_addr[6];     /* The own ETHERNET ID */
                                   /* overrides the ONBOARD */
                                   /* Ethernet ID */
   unsigned char   **multi_addr;   /* pointer to a list of */
                                   /* Multicast addresses */
   unsigned long   License;        /* the License number */
                                   /* of the board  */
   unsigned short  dma_modes;      /* DMA mode */
   unsigned long   a32_base;       /* base address of master board */
   unsigned long   a64_base;       /* high lword of master base if A64 */
};
```

Usage of L_mode with FDDI29:
Bridging Support:
For usage of the FDDI29 in bridging applications some configuration bits of the FDDI controller chipset have been made available at the users interface. Setting of these bits should be handled with great care. For detailed information of the meaning of these bits one should refer to the NS FDDI controller manuals.

- 0x0010 - set BOSEL bit of MAC MCMR2 register
- 0x0020 - set VST bit of BSI R1CR0 register (tx channel 1)
- 0x0040 - set SAT bit of BSI R1CR0 register (tx channel 1)

- 0x0080 - set FCT bit of BSI R1CR0 register (tx channel 1)

Transmit channel 0 ist reserved for SMT private usage and can not be manipulated.

Usage of L_mode with ETH29:

- 0x8000 - set controller to promiscous mode

DMA Support with FDDI29:
The new FDDI29 VMEbus board is capable of doing DMA transfers via the VMEbus when equipped with the SCV64 VMEbus Controller (DMA option). To make use of this option the fields *dma_modes, a32_base, a64_base* must be supported.
The the field `dma_modes` controls the behavior of the DMA of the FDDI29. The following bits are currently defined:

- 0x0000 - normal DMA mode - no BLT, no MBLT
- 0x0001 - `BLT: use` block mode transfer for DMA
- 0x0002 `- MBLT` : use multiplexed (D64) block mode transfers for DMA
- 0x0004 - NOREL: don´t release the bus during BLT or MBLT transfers
- 0x0008 - NOMIN: ignore minimum frame size for DMA transfers

The DMA controller will give up the bus at least every 8us or when the DMA Fifo in the SCV64 underruns to give other busmasters a chance to aquire the bus. This may lead to problems with some VMEbus controllers or boards. Therefore the controller can be forced by setting NOREL not to give up the bus until the complete DMA transfer has been finished.
When using the NOREL option, the VMEbus timeout must be large enough to allow DMA transfers with block sizes up to 4096 bytes.
The bit NOMIN forces the controller to do DMA transfers even with small frames. Without setting this bit all tranfers of less the 256 byte are directly done by the CPU. Setting the Bit will lead to better VMEbus utilization for the cost of a higher CPU load.
The field `a32_base` contains the base address of the master board as seen by the slave board via the VMEbus. If DMA should not be used, this field must be initialized to 0. If operating in A64 mode, the field `a64_base` contains the high lword of the master base address; however, this field is not supported yet. (for further information see Chapter "L2 DMA Commands).

L2_STOP

The command **L2_STOP** is used to stop operation at layer 2. No parameters are needed.

L2_ATTP

A communication port for a new protocol is opened on the slaveboard by the command **L2_ATTP** (attach protocol). The protocols are being distinguished by means of the used Ethernet types (one or more per protocol).
L2_ATTP needs following parameter:

- 0x0080 - set FCT bit of BSI R1CR0 register (tx channel 1)

Transmit channel 0 ist reserved for SMT private usage and can not be manipulated.

Usage of L_mode with ETH29:

- 0x8000 - set controller to promiscous mode

DMA Support with FDDI29:
The new FDDI29 VMEbus board is capable of doing DMA transfers via the VMEbus when equipped with the SCV64 VMEbus Controller (DMA option). To make use of this option the fields *dma_modes, a32_base, a64_base* must be supported.
The the field `dma_modes` controls the behavior of the DMA of the FDDI29. The following bits are currently defined:

- 0x0000 - normal DMA mode - no BLT, no MBLT
- 0x0001 - `BLT: use` block mode transfer for DMA
- 0x0002 `- MBLT` : use multiplexed (D64) block mode transfers for DMA
- 0x0004 - NOREL: don´t release the bus during BLT or MBLT transfers
- 0x0008 - NOMIN: ignore minimum frame size for DMA transfers

The DMA controller will give up the bus at least every 8us or when the DMA Fifo in the SCV64 underruns to give other busmasters a chance to aquire the bus. This may lead to problems with some VMEbus controllers or boards. Therefore the controller can be forced by setting NOREL not to give up the bus until the complete DMA transfer has been finished.
When using the NOREL option, the VMEbus timeout must be large enough to allow DMA transfers with block sizes up to 4096 bytes.
The bit NOMIN forces the controller to do DMA transfers even with small frames. Without setting this bit all tranfers of less the 256 byte are directly done by the CPU. Setting the Bit will lead to better VMEbus utilization for the cost of a higher CPU load.
The field `a32_base` contains the base address of the master board as seen by the slave board via the VMEbus. If DMA should not be used, this field must be initialized to 0. If operating in A64 mode, the field `a64_base` contains the high lword of the master base address; however, this field is not supported yet. (for further information see Chapter "L2 DMA Commands).

L2_STOP

The command **L2_STOP** is used to stop operation at layer 2. No parameters are needed.

L2_ATTP

A communication port for a new protocol is opened on the slaveboard by the command **L2_ATTP** (attach protocol). The protocols are being distinguished by means of the used Ethernet types (one or more per protocol).
L2_ATTP needs following parameter:

```
struct L2C_attp_p {                 /* Attach Proto Structure */
   long           proto_id;        /* protocol ID number */
   long           (*ifcall)();     /* interface-function */
                                   /* to be called from L2 */
   long           (*dispcall)();   /* Dispatcher of high level */
                                   /* protocoll */
   long           a6stat           /* static storage pointer */
                                   /* to be used for all */
                                   /* upcalls */
   unsigned short type;            /* The Ethernet type to */
                                   /* be assigned to */
};
```

The elements **ifcall, dispcall, a6stat** are used by an intelligent slaveboard only for internal calls and they have to be initialisated to 0. When a new protocol is supposed to be installed, the desired Ethernet type is recorded in **type**. The field **proto_id** must have one of following values when first requested:

- (0..MAXPROTO-1) - use a allready known or predefined ("well known") protocol ID
- (-1) - slaveboard will attach a new number for this protocol

After the completion of the command the new allocated protocol number is written in **proto_id**. The value of MAXPROTO is 8 in the current implementations.

If another Ethernet type shall be bound to an already existing protocol, **L2_ATTP** is called up again, the desired Ethernet typ is entered in **type** and the already existing port number is entered in **port**. If this port is desired to receive all incomming frames (for bridging or routing applications), **type** should be set to **(-1)**;
All frames which are received on this port are delivered to the host CPU by the Action-Slave-to-Host call **L2_RCV**.

For intelligent slaveboards with onboard firmware the protocol ports are attached by the firmware during the start-up. These protocols get "well known" port numbers, which can be used by other calls (for example **L2_VECTOR** (see below)) to access this port.

```
#define COMPROT_OS9          4       /* OS9Net */
#define COMPROT_TCPIP        5       /* TCP/IP */
#define COMPROT_DECNET       6       /* DECNet */
#define  COMPROT_ISO         7       /* ISO-OSI */
```

L2_ADDTYPE

The command **L2_ADDTYPE** is used to add an ethernet type to be recogniced to a previously opened port (see L2_ATTP)
L2_ADDTYPE needs the following parameters:

```
struct L2C_addt {
   long           proto_id;        /* protocol ID number */
   unsigned short type;            /* The Ethernet type to be recogniced*/
}
```

```
struct L2C_attp_p {                /* Attach Proto Structure */
   long           proto_id;        /* protocol ID number */
   long           (*ifcall)();     /* interface-function */
                                   /* to be called from L2 */
   long           (*dispcall)();   /* Dispatcher of high level */
                                   /* protocoll */
   long           a6stat           /* static storage pointer */
                                   /* to be used for all */
                                   /* upcalls */
   unsigned short  type;           /* The Ethernet type to */
                                   /* be assigned to */
};
```

The elements **ifcall, dispcall, a6stat** are used by an intelligent slaveboard only for internal calls and they have to be initialisated to 0. When a new protocol is supposed to be installed, the desired Ethernet type is recorded in **type**. The field **proto_id** must have one of following values when first requested:

- (0..MAXPROTO-1) - use a allready known or predefined ("well known") protocol ID
- (-1) - slaveboard will attach a new number for this protocol

After the completion of the command the new allocated protocol number is written in **proto_id**. The value of MAXPROTO is 8 in the current implementations.

If another Ethernet type shall be bound to an already existing protocol, **L2_ATTP** is called up again, the desired Ethernet typ is entered in **type** and the already existing port number is entered in **port**. If this port is desired to receive all incomming frames (for bridging or routing applications), **type** should be set to **(-1)**;
All frames which are received on this port are delivered to the host CPU by the Action-Slave-to-Host call **L2_RCV**.

For intelligent slaveboards with onboard firmware the protocol ports are attached by the firmware during the start-up. These protocols get "well known" port numbers, which can be used by other calls (for example **L2_VECTOR** (see below)) to access this port.

```
#define COMPROT_OS9          4      /* OS9Net */
#define COMPROT_TCPIP        5      /* TCP/IP */
#define COMPROT_DECNET       6      /* DECNet */
#define  COMPROT_ISO         7      /* ISO-OSI */
```

L2_ADDTYPE

The command **L2_ADDTYPE** is used to add an ethernet type to be recogniced to a previously opened port (see L2_ATTP)
L2_ADDTYPE needs the following parameters:

```
struct L2C_addt {
   long           proto_id;        /* protocol ID number */
   unsigned short  type;           /* The Ethernet type to be recogniced*/
}
```

L2_DETACH

The command **L2_DETACH** detaches a protocol that has previously been attached.
L2_DETACH needs the following parameter:

```
struct L2C_attp_p {               /* Attach Proto Structure */
   long           proto_id;       /* protocol ID number */
   long           (*ifcall)();    /* interface-function */
                                  /* to be called from L2 */
   long           (*dispcall)();  /* Dispatcher of high level */
                                  /* protocoll */
   long           a6stat          /* static storage pointer */
                                  /* to be used for all */
                                  /* upcalls */
   unsigned short  type;          /* The Ethernet type to */
                                  /* be assigned to */
};
```

This is the same structure used for **L2_ATTP**. For the command **L2_DETACH**, only the member variable *proto_id* of the structure is used to identify which protocol to detach. For further information about protocol ID's see the documentation about **L2_ATTP** above.

L2_VECTOR

An interrupt vector and level is being defined for an existing port (protocol) by the command **L2_VECTOR**. They are used for all **ActS2H**-commands, which generate a VMEbus interrupt. The **L2_VECTOR** needs the following parameter:

```
struct L2C_addvect {
          u_long port             /* the number of the port */
          u_char board_vector     /* VMEbus vector */
          u_char irq_level        /* and level to be assigned */
                                  /* to the port */
};
```

Different protocols may use different interrupt vectors and levels. Vector and level are mapped out only for the port number.

L2_ETHID

The 6 digit IEEE identification stored on the Ethernet board can be read back by the command **L2_ETHID**. N.A.T. GmbH gives out a unique address for every Ethernet board from her address pool (00-40-42-...), allocated by the IEEE Standard Office.
The command reads back the Ethernet address in the reserved cells "ethid":

```
u_char ethid[6];
```

Attention: L2_INIT, L2_ATTP, L2_VECTOR, L2_ETHID generate no VMEbus interrupt for the transmission of the return values, which means that the **ActS2H** cell has to be polled

L2_DETACH

The command **L2_DETACH** detaches a protocol that has previously been attached.
L2_DETACH needs the following parameter:

```
struct L2C_attp_p {                 /* Attach Proto Structure */
   long           proto_id;         /* protocol ID number */
   long           (*ifcall)();      /* interface-function */
                                    /* to be called from L2 */
   long           (*dispcall)();    /* Dispatcher of high level */
                                    /* protocoll */
   long           a6stat            /* static storage pointer */
                                    /* to be used for all */
                                    /* upcalls */
   unsigned short  type;            /* The Ethernet type to */
                                    /* be assigned to */
};
```

This is the same structure used for **L2_ATTP**. For the command **L2_DETACH**, only the member variable *proto_id* of the structure is used to identify which protocol to detach. For further information about protocol ID's see the documentation about **L2_ATTP** above.

L2_VECTOR

An interrupt vector and level is being defined for an existing port (protocol) by the command **L2_VECTOR**. They are used for all **ActS2H**-commands, which generate a VMEbus interrupt. The **L2_VECTOR** needs the following parameter:

```
struct L2C_addvect {
          u_long port              /* the number of the port */
          u_char board_vector      /* VMEbus vector */
          u_char irq_level         /* and level to be assigned */
                                   /* to the port */
};
```

Different protocols may use different interrupt vectors and levels. Vector and level are mapped out only for the port number.

L2_ETHID

The 6 digit IEEE identification stored on the Ethernet board can be read back by the command **L2_ETHID**. N.A.T. GmbH gives out a unique address for every Ethernet board from her address pool (00-40-42-...), allocated by the IEEE Standard Office.
The command reads back the Ethernet address in the reserved cells "ethid":

```
u_char ethid[6];
```

Attention: L2_INIT, L2_ATTP, L2_VECTOR, L2_ETHID generate no VMEbus interrupt for the transmission of the return values, which means that the **ActS2H** cell has to be polled

until it has a value unequal 0.
Background: These routines are usually requested by the driver init routine at a time when the interrupt handler of the masterboard has not been installed yet.

L2_PHYS_ETHID

The command **L2_PHYS_ETHID** reads back the physical Ethernet ID of the board. The command reads back the Ethernet address in the reserved cells "ethid":

```
u_char ethid[6];
```

Usually, the logical and the physical Ethernet ID are the same. They only differ if you have specified a logical Ethernet ID different from the physical ID.

L2_LOG_ETHID

The command **L2_LOG_ETHID** reads back the logical Ethernet ID of the board. The command reads back the Ethernet address in the reserved cells "ethid":

```
u_char ethid[6];
```

Usually, the logical and the physical Ethernet ID are the same. They only differ if you have specified a logical Ethernet ID different from the physical ID.

MEMINFO

The command **MEMINFO** returns information about the memory structure of the slave board. **MEMINFO** returns the following structure :

```
struct meminfo {
  ULONG   sys,          /* current available system memory */
          mbuf,         /* current available mbuf memory */
          page,         /* current available page memory */
          proc;         /* current available process descriptor cache memory */
};
```

TASKINFO

The command **TASKINFO** returns information about the currently running processes on the slave board. **TASKINFO** returns an array of the following structure :

```
struct taskinfo {
  USHORT  id,           /* process ID */
          state;        /* process state */
  ULONG   pc;           /* current program counter */
};
```

The mbuf carrying the information contains up to 12 structures. If less are used, the last structure contains pc = 0.

until it has a value unequal 0.
Background: These routines are usually requested by the driver init routine at a time when the interrupt handler of the masterboard has not been installed yet.

L2_PHYS_ETHID

The command **L2_PHYS_ETHID** reads back the physical Ethernet ID of the board. The command reads back the Ethernet address in the reserved cells "ethid":

```
u_char ethid[6];
```

Usually, the logical and the physical Ethernet ID are the same. They only differ if you have specified a logical Ethernet ID different from the physical ID.

L2_LOG_ETHID

The command **L2_LOG_ETHID** reads back the logical Ethernet ID of the board. The command reads back the Ethernet address in the reserved cells "ethid":

```
u_char ethid[6];
```

Usually, the logical and the physical Ethernet ID are the same. They only differ if you have specified a logical Ethernet ID different from the physical ID.

MEMINFO

The command **MEMINFO** returns information about the memory structure of the slave board. **MEMINFO** returns the following structure :

```
struct meminfo {
  ULONG   sys,            /* current available system memory */
          mbuf,           /* current available mbuf memory */
          page,           /* current available page memory */
          proc;           /* current available process descriptor cache memory */
};
```

TASKINFO

The command **TASKINFO** returns information about the currently running processes on the slave board. **TASKINFO** returns an array of the following structure :

```
struct taskinfo {
  USHORT  id,             /* process ID */
          state;          /* process state */
  ULONG   pc;             /* current program counter */
};
```

The mbuf carrying the information contains up to 12 structures. If less are used, the last structure contains pc = 0.

3.1.2 LAYER2 - Action Commands - data transfer via raw interface

L2_SEND

The command **L2_SEND** serves for sending a data package. The datas have to be written in a page, which had to be ordered from the slaveboard through **pget** (see above).
L2_SEND needs the following parameters:

```
struct L2_send {
   u_char  *addr;          /* address of page containing Data
   u_long  len;            /* total frame length
                           /* (including Ethernet header)
   u_char  ethid[6];       /* destination node ethernet ID
   u_short type;           /* Ethernet type to be used
};
```

The Ethernet header is added by Layer 2; therefore a corresponding space has to be reserved in the data area of the package. (see **struct llc_header** in l2.h too)
Layer 2 releases the used page and the mbuf.
L2_Send gives back no confirmation to the masterboard about the success of the action. The security of the datas has to be ensured by higher protocol levels.

L2_RCV

The command **L2_RCV** is send from the slaveboard to the masterboard (**ActS2H**), when a package for an open port has been received.
At the same time a VMEbus interrupt with the vector belonging to this port is generated.
The parameter of the command **L2_RCV** are transmitted by following structure:

```
struct BCB {
   u_char   *b_addr;         /* Address of frame on Slaveboard */
   short    b_len;           /* size of the frame buffer */
   short    b_msglen;        /* size of data */
};
```

The pointer "*b_addr" contains the address of a page (on the slaveboard) in which are written the frame datas including the Ethernet header. After the treatment, the page has to be released through **pput**. The command **L2_RCV** should be received on the masterboard in the interrupt routine to release the **ActS2H** cell as soon as possible.

3.1.3 LAYER 2 - Action Commands - CAM Support

The FDDI29 Board can be equipped optionaly with a CAM module. This CAM can be programmed by the interface call **L2_CAMCMD** . The parameters of the command **L2_CAMCMD** have the following structure:

3.1.2 LAYER2 - Action Commands - data transfer via raw interface

L2_SEND

The command **L2_SEND** serves for sending a data package. The datas have to be written in a page, which had to be ordered from the slaveboard through **pget** (see above).
L2_SEND needs the following parameters:

```
struct L2_send {
   u_char  *addr;          /* address of page containing Data
   u_long  len;            /* total frame length
                           /* (including Ethernet header)
   u_char  ethid[6];       /* destination node ethernet ID
   u_short type;           /* Ethernet type to be used
};
```

The Ethernet header is added by Layer 2; therefore a corresponding space has to be reserved in the data area of the package. (see **struct llc_header** in l2.h too)
Layer 2 releases the used page and the mbuf.
L2_Send gives back no confirmation to the masterboard about the success of the action. The security of the datas has to be ensured by higher protocol levels.

L2_RCV

The command **L2_RCV** is send from the slaveboard to the masterboard (**ActS2H**), when a package for an open port has been received.
At the same time a VMEbus interrupt with the vector belonging to this port is generated.
The parameter of the command **L2_RCV** are transmitted by following structure:

```
struct BCB {
   u_char   *b_addr;         /* Address of frame on Slaveboard */
   short    b_len;           /* size of the frame buffer */
   short    b_msglen;        /* size of data */
};
```

The pointer "*b_addr" contains the address of a page (on the slaveboard) in which are written the frame datas including the Ethernet header. After the treatment, the page has to be released through **pput**. The command **L2_RCV** should be received on the masterboard in the interrupt routine to release the **ActS2H** cell as soon as possible.

3.1.3 LAYER 2 - Action Commands - CAM Support

The FDDI29 Board can be equipped optionaly with a CAM module. This CAM can be programmed by the interface call **L2_CAMCMD** . The parameters of the command **L2_CAMCMD** have the following structure:

```
struct Par_CamCmd {
   u_short commands[42];                  /* commands to the cam */
   u_short result[8];                     /* result area for read commands */
}
```

The parameters of the **L2_CamCmd** action is a series of (u_short) command words which are written to the individual registers of the CAM module. The command words must be arranged in a table of the following form:

```
u_short commands [] = {
        reg_addr1, command1,      /* first command to CAM */
        reg_addr2, command2,      /* second command to CAM */
        ....         ....
        0xffff                    /* End of table identifier */
}
```

This table must be placed in the commands area of the **L2_CamCmd** mbuf. Possible result values from these commands may be optained in the results area.
The register address in the command table allows access to any register on the CAM-Module (SRT, FIFO, Control and Status register). To build the required addresses only the lower 10 bits of (u_short) address field are used. The unused upper 6 Bits in the address field have some special codings:

```
0x0XYZ - write command or data to CAM register XYZ (word access)
0x1XYZ - read data from CAM register XYZ (word access)
0x2abc - pseudo command
0x3XYZ - write byte to CAM control registers (0x300...)
0x4XYZ - read byte from CAM control resgisters (0x300...)
0x5XYZ - write byte to FDDI Chip set
0x6XYZ - read byte from FDDI Chip set
```

For access commands to the FDDI chip set, the individual addresses of the chips must be known. The internal address decoding of the FDDI29 board is as follows:

```
0x0000 - MACSI - MAC
0x0100 - MACSI - BSI
0x0200 - CAM Module
0x0400 - Player A and SAS
0x0600 - Player B
```

For a detailed description of the FDDI chip set a the register usage please refer to the National Semiconductor FDDI Handbook.
WARNING: as the manipulation of the FDDI chip set by these commands is possible it should be done with great care. Writing a wrong value in one of these register may corrupt the hole FDDI ring, as the chip set normaly is under control of the onboard SMT software.

The entries for a read cycle in the command area have the following form:

```
struct Par_CamCmd {
  u_short commands[42];                 /* commands to the cam */
  u_short result[8];                    /* result area for read commands */
}
```

The parameters of the **L2_CamCmd** action is a series of (u_short) command words which are written to the individual registers of the CAM module. The command words must be arranged in a table of the following form:

```
u_short commands [] = {
        reg_addr1, command1,      /* first command to CAM */
        reg_addr2, command2,      /* second command to CAM */
        ....         ....
        0xffff                    /* End of table identifier */
}
```

This table must be placed in the commands area of the **L2_CamCmd** mbuf. Possible result values from these commands may be optained in the results area.
The register address in the command table allows access to any register on the CAM-Module (SRT, FIFO, Control and Status register). To build the required addresses only the lower 10 bits of (u_short) address field are used. The unused upper 6 Bits in the address field have some special codings:

```
0x0XYZ - write command or data to CAM register XYZ (word access)
0x1XYZ - read data from CAM register XYZ (word access)
0x2abc - pseudo command
0x3XYZ - write byte to CAM control registers (0x300...)
0x4XYZ - read byte from CAM control resgisters (0x300...)
0x5XYZ - write byte to FDDI Chip set
0x6XYZ - read byte from FDDI Chip set
```

For access commands to the FDDI chip set, the individual addresses of the chips must be known. The internal address decoding of the FDDI29 board is as follows:

```
0x0000 - MACSI - MAC
0x0100 - MACSI - BSI
0x0200 - CAM Module
0x0400 - Player A and SAS
0x0600 - Player B
```

For a detailed description of the FDDI chip set a the register usage please refer to the National Semiconductor FDDI Handbook.
WARNING: as the manipulation of the FDDI chip set by these commands is possible it should be done with great care. Writing a wrong value in one of these register may corrupt the hole FDDI ring, as the chip set normaly is under control of the onboard SMT software.

The entries for a read cycle in the command area have the following form:

```
u_short commands [] = {
        ....  ,      ....
        0x1XYZ,      count,       /* read command, read count no. */
        ....  ,      .....        /* of values from register */
}
```

The result values are placed in the result area of the mbuf.

The following pseudo commands are defined:

```
  SLEEP: 0x2000,     ticks        /* sleep for number of ticks */
```

This command is useful when data has to be read from the CAM memory by one of the CAM routines, because it may take some time until the routine can be executed and the result is valid (due to currently running address filtering).

3.2 LAYER 2 - DMA Support

To support the DMA facility on the new FDDI29 board the the command L2_SEND_DMA, L2_RCV_DMA and L2_ADD_DMA_BUFFERS have been added to the L2 interface.
In difference to the standard L2 send/receive interface all data buffers are kept in the main memory of the host CPU. The FDDI29 will access the data buffers by DMA transfer. The DMA commands use the following structures:

L2_SEND_DMA

The L2_SEND_DMA command is capable of sending a list of buffers as one frame. The first fragment must contain the FDDI-SNAP header including 4 FC bytes. The List of fragments is terminated by a Zero-Pointer. The number of fragments is limited to 8. To achieve optimal performance a frame should consist of as less fragments as possible and the fragment addresses should start on long word boundaries.

```
struct L2_send_dma {
   u_char  *addr_1;        /* address of first Data fragment
   u_long  len_1;          /* length of first data frament
   u_char  *addr_1;        /* address of second Data fragment
   u_long  len_1;          /* length of sencond data frament
   .....
   u_char  *addr_n;        /* address of n-th Data fragment (n(max)=8)
   u_long  len_n;          /* length of n-th data frament
   0;                      /* 0 terminates the list
};
```

All addresses must be local to the host CPU (without any VMEbus offset).
The L2_SEND_DMA returns to the host CPU by an normal Action S2H command when the frame has been processed.

```
u_short commands [] = {
        ....  ,     ....
        0x1XYZ,     count,       /* read command, read count no. */
        ....  ,     .....        /* of values from register */
}
```

The result values are placed in the result area of the mbuf.

The following pseudo commands are defined:

```
  SLEEP: 0x2000,    ticks        /* sleep for number of ticks */
```

This command is useful when data has to be read from the CAM memory by one of the CAM routines, because it may take some time until the routine can be executed and the result is valid (due to currently running address filtering).

3.2 LAYER 2 - DMA Support

To support the DMA facility on the new FDDI29 board the the command L2_SEND_DMA, L2_RCV_DMA and L2_ADD_DMA_BUFFERS have been added to the L2 interface.
In difference to the standard L2 send/receive interface all data buffers are kept in the main memory of the host CPU. The FDDI29 will access the data buffers by DMA transfer. The DMA commands use the following structures:

L2_SEND_DMA

The L2_SEND_DMA command is capable of sending a list of buffers as one frame. The first fragment must contain the FDDI-SNAP header including 4 FC bytes. The List of fragments is terminated by a Zero-Pointer. The number of fragments is limited to 8. To achieve optimal performance a frame should consist of as less fragments as possible and the fragment addresses should start on long word boundaries.

```
struct L2_send_dma {
   u_char  *addr_1;        /* address of first Data fragment
   u_long  len_1;          /* length of first data frament
   u_char  *addr_1;        /* address of second Data fragment
   u_long  len_1;          /* length of sencond data frament
   .....
   u_char  *addr_n;        /* address of n-th Data fragment (n(max)=8)
   u_long  len_n;          /* length of n-th data frament
   0;                      /* 0 terminates the list
};
```

All addresses must be local to the host CPU (without any VMEbus offset).
The L2_SEND_DMA returns to the host CPU by an normal Action S2H command when the frame has been processed.

L2_ADD_DMA_BUFFERS

With the L2_ADD_DMA_BUFFERS the host CPU hands over buffers into which the FDDI29 strores received frames. Like the L2_SEND_DMA this also ist a list of fragments. Each fragment needs not to be the full FDDI frames length of 4500 bytes. The FDDI29 is capable to split receive buffers among several fragments.

```
struct L2_add_dma_buf {
   u_char  *addr_1;        /* address of first Data fragment
   u_long  len_1;          /* length of first data frament
   u_char  *addr_1;        /* address of second Data fragment
   u_long  len_1;          /* length of sencond data frament
   .....
   u_char  *addr_n;        /* address of n-th Data fragment (n(max)= 11)
   u_long  len_n;          /* length of n-th data frament
   0;                      /* 0 terminates the list or n=14
};
```

L2_RCV_DMA

The Action S2H command L2_RCV_DMA informs the host CPU of the reception of an frame. The host should maintain the field dma_buf_left which contains the actual amount of buffer space (in bytes) for receiving frames on the slave board. There always should be left at least enough space for receiving one fddi frame (4500 bytes). If there is not enough space left on the slave board all incomming frames will be dropped. With the dma_left value the host CPU is capable to decide when to issue a new L2_ADD_DMA_BUFFERS command

```
struct L2_rcv_dma {
   u_long  frame_len;      /* total size of this frame
   u_long  dma_buf_left;   /* currently amount of buffer space for receiving frames
                           /* on the Sleave board
   u_char  *addr_1;        /* address of first Data fragment
   u_long  len_1;          /* length of first data frament
   u_char  *addr_1;        /* address of second Data fragment
   u_long  len_1;          /* length of sencond data frament
   .....
   u_char  *addr_n;        /* address of n-th Data fragment (n(max)=8)
   u_long  len_n;          /* length of n-th data frament
   0;                      /* 0 terminates the list
};
```

All addresses are local to the host CPU.

L2_ADD_DMA_BUFFERS

With the L2_ADD_DMA_BUFFERS the host CPU hands over buffers into which the FDDI29 strores received frames. Like the L2_SEND_DMA this also ist a list of fragments. Each fragment needs not to be the full FDDI frames length of 4500 bytes. The FDDI29 is capable to split receive buffers among several fragments.

```
struct L2_add_dma_buf {
   u_char  *addr_1;        /* address of first Data fragment
   u_long  len_1;          /* length of first data frament
   u_char  *addr_1;        /* address of second Data fragment
   u_long  len_1;          /* length of sencond data frament
   .....
   u_char  *addr_n;        /* address of n-th Data fragment (n(max)= 11)
   u_long  len_n;          /* length of n-th data frament
   0;                      /* 0 terminates the list or n=14
};
```

L2_RCV_DMA

The Action S2H command L2_RCV_DMA informs the host CPU of the reception of an frame. The host should maintain the field dma_buf_left which contains the actual amount of buffer space (in bytes) for receiving frames on the slave board. There always should be left at least enough space for receiving one fddi frame (4500 bytes). If there is not enough space left on the slave board all incomming frames will be dropped. With the dma_left value the host CPU is capable to decide when to issue a new L2_ADD_DMA_BUFFERS command

```
struct L2_rcv_dma {
   u_long  frame_len;      /* total size of this frame
   u_long  dma_buf_left;   /* currently amount of buffer space for receiving frames
                           /* on the Sleave board
   u_char  *addr_1;        /* address of first Data fragment
   u_long  len_1;          /* length of first data frament
   u_char  *addr_1;        /* address of second Data fragment
   u_long  len_1;          /* length of sencond data frament
   .....
   u_char  *addr_n;        /* address of n-th Data fragment (n(max)=8)
   u_long  len_n;          /* length of n-th data frament
   0;                      /* 0 terminates the list
};
```

All addresses are local to the host CPU.

4 Socket Level Interface

4.1 TCP/IP - Action Commands

The following structures contain elements, which correspond to the parameters concerning syntax and semantics, which are needed when the BSD socket interface is used. For further explanations we recommend to consider more detailed literature, for ex.:

- Internetworking with TCP/IP from Douglas Comer,
- UNIX Network Programming from W. R. Stevens,
- Design and Implementation of the 4.3BSD Unix Operating System from Leffler/ McKusick/Karels/Quaterman

or every Unix-manual, in which the socket-interface is being described.

SOC_SOCKET

For the command **SOC_SOCKET** following structure is used:

```
struct Par_Socket {       /* Socket-Call */
   int             dom;
   int             type;
   int             proto;
   unsigned long   uid;    /* ID of calling user */
};
```

The last parameter **uid** is additionally required for the standard-parameters, to check out if the access to priviledged sockets (for ex. for server-programs) is sufficiently authorized.

SOC_BIND

For the command **SOC_BIND** following structure is used:

```
struct Par_Bind {         /* Bind-Call */
   int             socket;
   unsigned char   *name;
   int             namelen;
};
```

4 Socket Level Interface

4.1 TCP/IP - Action Commands

The following structures contain elements, which correspond to the parameters concerning syntax and semantics, which are needed when the BSD socket interface is used. For further explanations we recommend to consider more detailed literature, for ex.:

- Internetworking with TCP/IP from Douglas Comer,
- UNIX Network Programming from W. R. Stevens,
- Design and Implementation of the 4.3BSD Unix Operating System from Leffler/ McKusick/Karels/Quaterman

or every Unix-manual, in which the socket-interface is being described.

SOC_SOCKET

For the command **SOC_SOCKET** following structure is used:

```
struct Par_Socket {       /* Socket-Call */
   int            dom;
   int            type;
   int            proto;
   unsigned long  uid;    /* ID of calling user */
};
```

The last parameter **uid** is additionally required for the standard-parameters, to check out if the access to priviledged sockets (for ex. for server-programs) is sufficiently authorized.

SOC_BIND

For the command **SOC_BIND** following structure is used:

```
struct Par_Bind {         /* Bind-Call */
   int            socket;
   unsigned char  *name;
   int            namelen;
};
```

SOC_LISTEN

For the command **SOC_LISTEN** following structure is used:

```
struct Par_Listen {       /* Listen-Call */
  int     socket;
  int     backlog;
};
```

SOC_ACCEPT

For the command **SOC_ACCEPT** the following structure is used:

```
struct Par_Accept {       /* Accept-Call */
  int             socket;
  unsigned char   *name;
  int             *anamelen;
};
```

SOC_CONNECT

For the command **SOC_CONNECT** the following structure is used:

```
struct Par_Connect {      /* Connect-Call */
  int             socket;
  unsigned char   *name;
  int             namelen;
};
```

SOC_SENDTO

For the command **SOC_SENDTO** the following structure is used:

```
struct Par_Sendto {       /* Sendto-Call */
  int             socket;
  unsigned char   *buf;
  int             len;
  int             flags;
  unsigned char   *to;
  int             tolen;
};
```

SOC_SEND

For the command **SOC_SEND** the following structure is used:

```
struct Par_Send {         /* Send-Call */
  int             socket;
  unsigned char   *buf;
  int             len;
  int             flags;
};
```

SOC_LISTEN

For the command **SOC_LISTEN** following structure is used:

```
struct Par_Listen {       /* Listen-Call */
   int     socket;
   int     backlog;
};
```

SOC_ACCEPT

For the command **SOC_ACCEPT** the following structure is used:

```
struct Par_Accept {       /* Accept-Call */
   int             socket;
   unsigned char   *name;
   int             *anamelen;
};
```

SOC_CONNECT

For the command **SOC_CONNECT** the following structure is used:

```
struct Par_Connect {      /* Connect-Call */
   int             socket;
   unsigned char   *name;
   int             namelen;
};
```

SOC_SENDTO

For the command **SOC_SENDTO** the following structure is used:

```
struct Par_Sendto {       /* Sendto-Call */
   int             socket;
   unsigned char   *buf;
   int             len;
   int             flags;
   unsigned char   *to;
   int             tolen;
};
```

SOC_SEND

For the command **SOC_SEND** the following structure is used:

```
struct Par_Send {         /* Send-Call */
   int             socket;
   unsigned char   *buf;
   int             len;
   int             flags;
};
```

SOC_SENDMSG

For the command **SOC_SENDMSG** the following structure is used:

```
struct Par_Sendmsg {       /* Sendmsg-Call */
   int              socket;
   unsigned char    *msg;
   int              flags;
};
```

SOC_RECVFROM

For the command **SOC_RECVFROM** the following structure is used:

```
struct Par_Recvfrom {      /* Receive-From-Call */
   int              socket;
   unsigned char    *buf;
   int              len;
   int              flags;
   unsigned char    *from;
   int              *fromlenaddr;
};
```

SOC_RECV

For the command **SOC_RECV** the following structure is used:

```
struct Par_Recv {          /* Receive-Call */
   int              socket;
   unsigned char    *buf;
   int              len;
   int              flags;
};
```

SOC_RECVMSG

For the command **SOC_RECVMSG** the following structure is used:

```
struct Par_Recvmsg {       /* Receive-Message-Call */
   int              socket;
   unsigned char    *msg;
   int              flags;
};
```

SOC_SHUTDOWN

For the command **SOC_SHUTDOWN** the following structure is used:

```
struct Par_Shutdown {      /* Shutdown-Call */
   int      socket;
   int      how;
};
```

SOC_SENDMSG

For the command **SOC_SENDMSG** the following structure is used:

```
struct Par_Sendmsg {        /* Sendmsg-Call */
   int              socket;
   unsigned char    *msg;
   int              flags;
};
```

SOC_RECVFROM

For the command **SOC_RECVFROM** the following structure is used:

```
struct Par_Recvfrom {       /* Receive-From-Call */
   int              socket;
   unsigned char    *buf;
   int              len;
   int              flags;
   unsigned char    *from;
   int              *fromlenaddr;
};
```

SOC_RECV

For the command **SOC_RECV** the following structure is used:

```
struct Par_Recv {           /* Receive-Call */
   int              socket;
   unsigned char    *buf;
   int              len;
   int              flags;
};
```

SOC_RECVMSG

For the command **SOC_RECVMSG** the following structure is used:

```
struct Par_Recvmsg {        /* Receive-Message-Call */
   int              socket;
   unsigned char    *msg;
   int              flags;
};
```

SOC_SHUTDOWN

For the command **SOC_SHUTDOWN** the following structure is used:

```
struct Par_Shutdown {       /* Shutdown-Call */
   int      socket;
   int      how;
};
```

SOC_SETSOCKOPT

For the command **SOC_SETSOCKOPT** the following structure is used:

```
struct Par_Setsockopt {      /* Set-Socketoption-Call */
   int               socket;
   int               level;
   int               name;
   unsigned char     *val;
   int               valsize;
};
```

SOC_GETSOCKOPT

For the command **SOC_GETSOCKOPT** the following structure is used:

```
struct Par_Getsockopt {    /* Get-Socketoption-Call */
   int              socket;
   int              level;
   int              name;
   unsigned char    *val;
   int              *avalsize;
};
```

SOC_GETSOCKNAME

For the command **SOC_GETSOCKNAME** the following structure is used:

```
struct Par_Getsockname {   /* Get-Socket-Name-Call */
   int              socket;
   unsigned char    *asa;
   int              *alen;
};
```

SOC_GETPEERNAME

For the command **SOC_GETPEERNAME** the following structure is used:

```
struct Par_Getpeername {   /* Getpeername-Call */
   int              socket;
   unsigned char    *asa;
   int              *alen;
};
```

SOC_IOCTL

For the command **SOC_IOCTL** the following structure is used:

```
struct Par_Ioctl {         /* I/O-Control-Call */
   int              socket;
   unsigned long    cmd;
   unsigned char    *cmarg;
};
```

SOC_SETSOCKOPT

For the command **SOC_SETSOCKOPT** the following structure is used:

```
struct Par_Setsockopt {      /* Set-Socketoption-Call */
   int              socket;
   int              level;
   int              name;
   unsigned char    *val;
   int              valsize;
};
```

SOC_GETSOCKOPT

For the command **SOC_GETSOCKOPT** the following structure is used:

```
struct Par_Getsockopt {    /* Get-Socketoption-Call */
   int              socket;
   int              level;
   int              name;
   unsigned char    *val;
   int              *avalsize;
};
```

SOC_GETSOCKNAME

For the command **SOC_GETSOCKNAME** the following structure is used:

```
struct Par_Getsockname {   /* Get-Socket-Name-Call */
   int              socket;
   unsigned char    *asa;
   int              *alen;
};
```

SOC_GETPEERNAME

For the command **SOC_GETPEERNAME** the following structure is used:

```
struct Par_Getpeername {   /* Getpeername-Call */
   int              socket;
   unsigned char    *asa;
   int              *alen;
};
```

SOC_IOCTL

For the command **SOC_IOCTL** the following structure is used:

```
struct Par_Ioctl {         /* I/O-Control-Call */
   int              socket;
   unsigned long    cmd;
   unsigned char    *cmarg;
};
```

SOC_CLOSE

For the command **SOC_CLOSE** the folowing structure is used:

```
struct Par_Close { /* Socket-Close-Call */
    int     socket;
};
```

SOC_SELECT

For the command **SOC_SELECT** the following structure is used:

```
struct Par_Select {         /* Select-Call Action */
    int             socket;
    unsigned char   *procd;
    int             flag;
    int             clear_or_set;
};
```

The function **select** has to be realized mainly on the masterboard. The slaveboard makes available by the command **SOC_SELECT** only a mechanism, which generates a VMEbus interrupt when for a given socket a certain event has ocurred. The flag **flag** states to which kind of event should be reacted (0 = Exeption event, 1 = Read event, 2 = Write event). The mechanism is started with **clear_or_set** = 1 and cancelled with **clear_or_set** = 0. The interaction with other I/O paths of the operating system has to be realized on the masterboard by corresponding software (see select.c). The field **procd** contains the address of the process descriptor of the process, which performs the **SOC_SELECT** command.

4.2 Additional TCP/IP - Action Commands

To keep things user friendly, we added three calls to our socket library. For detailed description, see TCP/IP Manual, Appendix B.

SOC_SENDX

For the command **SOC_SENDX** the following structure is used:

```
struct Par_Sendx {          /* Send-Call Action */
    int     socket;
    caddr_t buf;
    int     len;
    int     flags;
};
```

SOC_CLOSE

For the command **SOC_CLOSE** the folowing structure is used:

```
struct Par_Close { /* Socket-Close-Call */
   int     socket;
};
```

SOC_SELECT

For the command **SOC_SELECT** the following structure is used:

```
struct Par_Select {       /* Select-Call Action */
   int             socket;
   unsigned char   *procd;
   int             flag;
   int             clear_or_set;
};
```

The function **select** has to be realized mainly on the masterboard. The slaveboard makes available by the command **SOC_SELECT** only a mechanism, which generates a VMEbus interrupt when for a given socket a certain event has ocurred. The flag **flag** states to which kind of event should be reacted (0 = Exeption event, 1 = Read event, 2 = Write event). The mechanism is started with **clear_or_set** = 1 and cancelled with **clear_or_set** = 0. The interaction with other I/O paths of the operating system has to be realized on the masterboard by corresponding software (see select.c). The field **procd** contains the address of the process descriptor of the process, which performs the **SOC_SELECT** command.

4.2 Additional TCP/IP - Action Commands

To keep things user friendly, we added three calls to our socket library. For detailed description, see TCP/IP Manual, Appendix B.

SOC_SENDX

For the command **SOC_SENDX** the following structure is used:

```
struct Par_Sendx {        /* Send-Call Action */
   int     socket;
   caddr_t buf;
   int     len;
   int     flags;
};
```

SYS_REQMEM

For the command **SYS_REQMEM** the following structure is used:

```
struct sysreqmem {
  ULONG   memsize;
  UCHAR   *memadr;        /* actually contains the address */
};
```

SYS_RETMEM

For the command **SYS_RETMEM** the following structure is used:

```
struct sysretmem {
  ULONG   memsize;
  UCHAR   *memadr;
};
```

4.3 Commands from the slaveboard to the masterboard

The following structures are used for commands from the slaveboard to the masterboard. The data part of the mbuf is filled with the corresponding **nifpar** structure, whereby within the **Opt**-union of the nifpar structure the structure corresponding to the command is used.
Each command is initiated by a VMEbus interrupt.

Examples are available in the interrupt routine of isockdrv.a.

S_COPYIN, S_COPYOUT and S_COPYOUT_M

For the commands **S_COPYIN, S_COPYOUT** and **S_COPYOUT_M** the following structure is used:

```
struct S_CopyData {       /* Copy Data Action */
  unsigned char   *Cpto;
  unsigned char   *Cpfrom;
  long            Cplen;
  unsigned short  Cppid;          /* procID of initiating host process */
};
```

The field **Cplen** states how many bytes have to be copied from the address **Cpfrom** to the address **Cpto**. After the masterboard has carried out the copy process, the address of the mbuf, with which the command has been initiated, has to be written into **ActH2S** and a mailbox interrupt has to be generated with the code **IRQ_SIGNAL**.

SYS_REQMEM

For the command **SYS_REQMEM** the following structure is used:

```
struct sysreqmem {
   ULONG   memsize;
   UCHAR   *memadr;        /* actually contains the address */
};
```

SYS_RETMEM

For the command **SYS_RETMEM** the following structure is used:

```
struct sysretmem {
   ULONG   memsize;
   UCHAR   *memadr;
};
```

4.3 Commands from the slaveboard to the masterboard

The following structures are used for commands from the slaveboard to the masterboard. The data part of the mbuf is filled with the corresponding **nifpar** structure, whereby within the **Opt**-union of the nifpar structure the structure corresponding to the command is used.
Each command is initiated by a VMEbus interrupt.

Examples are available in the interrupt routine of isockdrv.a.

S_COPYIN, S_COPYOUT and S_COPYOUT_M

For the commands **S_COPYIN, S_COPYOUT** and **S_COPYOUT_M** the following structure is used:

```
struct S_CopyData {       /* Copy Data Action */
   unsigned char   *Cpto;
   unsigned char   *Cpfrom;
   long            Cplen;
   unsigned short  Cppid;          /* procID of initiating host process */
};
```

The field **Cplen** states how many bytes have to be copied from the address **Cpfrom** to the address **Cpto**. After the masterboard has carried out the copy process, the address of the mbuf, with which the command has been initiated, has to be written into **ActH2S** and a mailbox interrupt has to be generated with the code **IRQ_SIGNAL**.

Concerning the command **S_COPYOUT_M** the address Cpfrom is not a single buffer, but the pointer to a list of mbufs which are chained up within the field m_next. The addresses used for the chain up are the internal addresses of the slaveboard, which means that for the access of the masterboard they have to be corrected accordingly. (addr = addr & (MAXRAM-1) | SLA-VEBASE)

S_SELWAKEUP

For the command **S_SELWAKEUP** the following structure is used:

```
struct S_Selwakeup {      /* Select-Wake-up-Call */
   unsigned char   *procd;
   int             socket;
   int             which;
};
```

In the field **procd** is written the process descriptor address of the process, which has carried out the command **SOC_SELECT**. In **socket** is stated for which socket an event has occured and in **which** is stated which event has occured. (0 = Exception, 1 = Read, 2 = Write). The masterboard has to free-up the mbuf by **mput**. (see isockdrv.a)

Concerning the command **S_COPYOUT_M** the address Cpfrom is not a single buffer, but the pointer to a list of mbufs which are chained up within the field m_next. The addresses used for the chain up are the internal addresses of the slaveboard, which means that for the access of the masterboard they have to be corrected accordingly. (addr = addr & (MAXRAM-1) | SLAVEBASE)

S_SELWAKEUP

For the command **S_SELWAKEUP** the following structure is used:

```
struct S_Selwakeup {      /* Select-Wake-up-Call */
   unsigned char   *procd;
   int             socket;
   int             which;
};
```

In the field **procd** is written the process descriptor address of the process, which has carried out the command **SOC_SELECT**. In **socket** is stated for which socket an event has occured and in **which** is stated which event has occured. (0 = Exception, 1 = Read, 2 = Write). The masterboard has to free-up the mbuf by **mput**. (see isockdrv.a)

5 nif - Codes

5.1 Commands from the masterboard to the slaveboard

```
#define L2_INIT         1             /* L2 - Init */
```
corr. element of the Opt-Union: init = struct L2M_init_p

```
#define L2_ATTP         2             /* L2 - Attach Port */
```
corr. element of the Opt-Union: attp = struct L2C_attp_p

```
#define L2_SEND         3             /* L2 - Send Data */
```
corr. element of the Opt-Union: bcb = struct BCB

```
#define L2_ETHID        4             /* get boards Ethernet ID */
```
corr. element of the Opt-Union: ethid

```
#define L2_DEBUG        5             /* set layer 2 debug level */
```
corr. element of the Opt-Union: debuglevel

```
#define L2_DETACH       6             /* detach a port from operation */
```
corr. element of the Opt-Union: L2_attproto = struct L2_attproto

```
#define L2_STOP         7             /* stop Layer 2 operation */
```
no parameters

```
#define L2_VECTOR       8             /* send IRQ vector and IRQ level*/
                                      /* for a specified protocol to  */
                                      /* Layer 2 */
```
corr. element of the Opt-Union: addvect = struct L2C_addvect

```
#define L2_PHYS_ETHID  9              /* get the boards physical */
                                      /* Ethernet ID */
```
corr. element of the Opt-Union: ethid = char ethid[6]

```
#define L2_LOG_ETHID   10             /* get the boards logical  */
                                      /* Ethernet ID */
```
corr. element of the Opt-Union: ethid = char ethid[6]

```
#define SOC_INIT        11            /* Initialize TCP/IP Softw. */
```
no parameters

5 nif - Codes

5.1 Commands from the masterboard to the slaveboard

```
#define L2_INIT       1            /* L2 - Init */
```
corr. element of the Opt-Union: init = struct L2M_init_p

```
#define L2_ATTP       2            /* L2 - Attach Port */
```
corr. element of the Opt-Union: attp = struct L2C_attp_p

```
#define L2_SEND       3            /* L2 - Send Data */
```
corr. element of the Opt-Union: bcb = struct BCB

```
#define L2_ETHID      4            /* get boards Ethernet ID */
```
corr. element of the Opt-Union: ethid

```
#define L2_DEBUG      5            /* set layer 2 debug level */
```
corr. element of the Opt-Union: debuglevel

```
#define L2_DETACH     6            /* detach a port from operation */
```
corr. element of the Opt-Union: L2_attproto = struct L2_attproto

```
#define L2_STOP       7            /* stop Layer 2 operation */
```
no parameters

```
#define L2_VECTOR     8            /* send IRQ vector and IRQ level*/
                                   /* for a specified protocol to  */
                                   /* Layer 2 */
```
corr. element of the Opt-Union: addvect = struct L2C_addvect

```
#define L2_PHYS_ETHID  9           /* get the boards physical */
                                   /* Ethernet ID */
```
corr. element of the Opt-Union: ethid = char ethid[6]

```
#define L2_LOG_ETHID   10          /* get the boards logical  */
                                   /* Ethernet ID */
```
corr. element of the Opt-Union: ethid = char ethid[6]

```
#define SOC_INIT      11           /* Initialize TCP/IP Softw. */
```
no parameters

```
#define SOC_SOCKET    12          /* Socket call */
```
corr. element of the Opt-Union: struct Par_Socket

```
#define SOC_BIND      13          /* Bind call */
```
corr. element of the Opt-Union: struct Par_Bind

```
#define SOC_LISTEN    14          /* Listen call */
```
corr. element of the Opt-Union: struct Par_Listen

```
#define SOC_ACCEPT    15          /* Accept call */
```
corr. element of the Opt-Union: struct Par_Accept

```
#define SOC_CONNECT   16          /* Connect call */
```
corr. element of the Opt-Union: struct Par_Connect

```
#define SOC_SENDTO    17          /* Sendto call */
```
corr. element of the Opt-Union: struct Par_Sendto

```
#define SOC_SEND      18          /* Send call */
```
corr. element of the Opt-Union: struct Par_Send

```
#define SOC_SENDMSG   19          /* Sendmsg call */
```
corr. element of the Opt-Union: struct Par_Sendmsg

```
#define SOC_RECVFROM  20          /* Recvfrom call */
```
corr. element of the Opt-Union: struct Par_Recvfrom

```
#define SOC_RECV      21          /* Recv call */
```
corr. element of the Opt-Union: struct Par_Recv

```
#define SOC_RECVMSG   22          /* Recvmsg call */
```
corr. element of the Opt-Union: struct Par_Recvmsg

```
#define SOC_SHUTDOWN  23          /* Shutdown call */
```
corr. element of the Opt-Union: struct Par_Shutdown

```
#define SOC_GETSOCKOPT      24    /* Getsockopt call */
```
corr. element of the Opt-Union: struct Par_Getsockopt

```
#define SOC_SETSOCKOPT      25    /* Setsockopt call */
```
corr. element of the Opt-Union: struct Par_Setsockopt

```
#define SOC_GETSOCKNAME     26    /* Getsockname call */
```
corr. element of the Opt-Union: struct Par_Getsockname

```
#define SOC_GETPEERNAME     27    /* Getpeername call */
```
corr. element of the Opt-Union: struct Par_Getpeername

```
#define SOC_SOCKET    12           /* Socket call */
```
corr. element of the Opt-Union: struct Par_Socket

```
#define SOC_BIND      13           /* Bind call */
```
corr. element of the Opt-Union: struct Par_Bind

```
#define SOC_LISTEN    14           /* Listen call */
```
corr. element of the Opt-Union: struct Par_Listen

```
#define SOC_ACCEPT    15           /* Accept call */
```
corr. element of the Opt-Union: struct Par_Accept

```
#define SOC_CONNECT   16           /* Connect call */
```
corr. element of the Opt-Union: struct Par_Connect

```
#define SOC_SENDTO    17           /* Sendto call */
```
corr. element of the Opt-Union: struct Par_Sendto

```
#define SOC_SEND      18           /* Send call */
```
corr. element of the Opt-Union: struct Par_Send

```
#define SOC_SENDMSG   19           /* Sendmsg call */
```
corr. element of the Opt-Union: struct Par_Sendmsg

```
#define SOC_RECVFROM  20           /* Recvfrom call */
```
corr. element of the Opt-Union: struct Par_Recvfrom

```
#define SOC_RECV      21           /* Recv call */
```
corr. element of the Opt-Union: struct Par_Recv

```
#define SOC_RECVMSG   22           /* Recvmsg call */
```
corr. element of the Opt-Union: struct Par_Recvmsg

```
#define SOC_SHUTDOWN  23           /* Shutdown call */
```
corr. element of the Opt-Union: struct Par_Shutdown

```
#define SOC_GETSOCKOPT      24     /* Getsockopt call */
```
corr. element of the Opt-Union: struct Par_Getsockopt

```
#define SOC_SETSOCKOPT      25     /* Setsockopt call */
```
corr. element of the Opt-Union: struct Par_Setsockopt

```
#define SOC_GETSOCKNAME     26     /* Getsockname call */
```
corr. element of the Opt-Union: struct Par_Getsockname

```
#define SOC_GETPEERNAME     27     /* Getpeername call */
```
corr. element of the Opt-Union: struct Par_Getpeername

```
#define SOC_IOCTL     28            /* Ioctl call for sockets */
```
corr. element of the Opt-Union: struct Par_Ioctl

```
#define SOC_CLOSE     29            /* Close call for sockets */
```
corr. element of the Opt-Union: struct Par_Close

```
#define SOC_SELECT    30            /* Select call for sockets */
```
corr. element of the Opt-Union: struct Par_Select

```
#define SOC_TEST_LICENSE    34      /* Test license number */
```
no parameters

```
#define MEMINFO             35      /* get Kernel memory info */
```
corr. element of the Opt-Union: meminfo = struct meminfo

```
#define L2_ATTPROTO         37      /* Attach protocol */
```
corr. element of the Opt-Union: L2_attproto = struct L2_attproto

```
#define TASKINFO            38      /* get process table info */
```
corr. element of the Opt-Union: taskinfo = struct taskinfo

```
#define SYS_REQMEM          39      /* get system memory */
```
corr. element of the Opt-Union: sysreqmem = struct sysreqmem

```
#define SYS_RETMEM          40      /* return allocated system mem */
```
corr. element of the Opt-Union: sysretmem = struct sysretmem

```
#define SOC_SENDX           41      /* special send call */
```
corr. element of the Opt-Union: Par_Sendx = struct Par_Sendx
```
#define L2_CAMCMD           42      /* CAm interface support call */
#define L2_CAMCMD_NOIRQ     43      /* CAMCMD using polling */
```



```
#define SOC_IOCTL     28           /* Ioctl call for sockets */
```
corr. element of the Opt-Union: struct Par_Ioctl

```
#define SOC_CLOSE     29           /* Close call for sockets */
```
corr. element of the Opt-Union: struct Par_Close

```
#define SOC_SELECT    30           /* Select call for sockets */
```
corr. element of the Opt-Union: struct Par_Select

```
#define SOC_TEST_LICENSE    34     /* Test license number */
```
no parameters

```
#define MEMINFO             35     /* get Kernel memory info */
```
corr. element of the Opt-Union: meminfo = struct meminfo

```
#define L2_ATTPROTO         37     /* Attach protocol */
```
corr. element of the Opt-Union: L2_attproto = struct L2_attproto

```
#define TASKINFO            38     /* get process table info */
```
corr. element of the Opt-Union: taskinfo = struct taskinfo

```
#define SYS_REQMEM          39     /* get system memory */
```
corr. element of the Opt-Union: sysreqmem = struct sysreqmem

```
#define SYS_RETMEM          40     /* return allocated system mem */
```
corr. element of the Opt-Union: sysretmem = struct sysretmem

```
#define SOC_SENDX           41     /* special send call */
```
corr. element of the Opt-Union: Par_Sendx = struct Par_Sendx
```
#define L2_CAMCMD           42     /* CAm interface support call */
#define L2_CAMCMD_NOIRQ     43     /* CAMCMD using polling */
```

5.2 Commands from the slaveboard to the masterboard

```
#define L2_RCV              101        /* L2 S2H - receive data */
#define S_COPYIN            102        /* Copy from masterboard */
                                       /* to slaveboard */

#define S_COPYOUT           103        /* Copy from slaveboard */
                                       /* to masterboard */

#define S_SELWAKEUP         104        /* Wakeup for select() */

#define S_COPYOUT_M         105        /* copy whole mbuf chain */
```

5.3 MBox-IRQ - Codes

The following codes have to be written into the mailbox cell of the slaveboard to generate the corresponding interrupt.
(see nif_a.a, nif.h).

```
#define IRQ_MGET      1      /* get an mbuf from slave */
#define IRQ_MPUT      2      /* put mbuf back to slave */
#define IRQ_ActH2S    3      /* Action-Host-to-Slave */
#define IRQ_ActS2H    4      /* Action-Slave-to-Host */
#define IRQ_PGET      5      /* get page */
#define IRQ_PPUT      6      /* put page */
#define IRQ_SNDRAW    7      /* send raw data */
#define IRQ_RCVRAW    8      /* receive raw data */
#define IRQ_SIGNAL    9      /* wake up slave process */
```

5.2 Commands from the slaveboard to the masterboard

```
#define L2_RCV              101         /* L2 S2H - receive data */
#define S_COPYIN            102         /* Copy from masterboard */
                                        /* to slaveboard */

#define S_COPYOUT           103         /* Copy from slaveboard */
                                        /* to masterboard */

#define S_SELWAKEUP         104         /* Wakeup for select() */

#define S_COPYOUT_M         105         /* copy whole mbuf chain */
```

5.3 MBox-IRQ - Codes

The following codes have to be written into the mailbox cell of the slaveboard to generate the corresponding interrupt.
(see nif_a.a, nif.h).

```
#define IRQ_MGET      1      /* get an mbuf from slave */
#define IRQ_MPUT      2      /* put mbuf back to slave */
#define IRQ_ActH2S    3      /* Action-Host-to-Slave */
#define IRQ_ActS2H    4      /* Action-Slave-to-Host */
#define IRQ_PGET      5      /* get page */
#define IRQ_PPUT      6      /* put page */
#define IRQ_SNDRAW    7      /* send raw data */
#define IRQ_RCVRAW    8      /* receive raw data */
#define IRQ_SIGNAL    9      /* wake up slave process */
```

6 Firmware Description: OK1-29K

All N.A.T. network boards are delivered with the multi-tasking kernel OK1 (Open Kernel 1). OK1 supplies all of the operating system resources required by the network software. In detail, it handles the:

- memory management
- time management
- signal handling
- task scheduling
- interrupt dispatching

One of OK1´s most important characteristics is its extremely short task switching times. This is a prerequisite for high speed handling of network protocols. OK1 is currently available for CPU cards based on the Motorola 680xy and AMD 29K.

6.1 Power Up Tests

The firmware tests all onboard components whenever the power is cycled or the reset switch is depressed. The test results are written to the “nif” field at the address = board´s base address + $4000.

The VMEbus line, SYSFAIL, will be driven accordingly: while the board is doing a reset, SYSFAIL will be active; after the reset and self-tests are completed, SYSFAIL will be inactive unless a failure has been detected.

The individual cells of the **“nif”** fields are assigned as follows:

Cell	Assignment
$4000	Status
$4004	Test Phase
$4008	Base Address RAM1
$400C	Maximum Address RAM1
$4010	Base Address RAM2
$4014	Maximum Address RAM2

The ***status*** cells can take the following values:

Value	Meaning
$5A5A 1234	Test in progress
$5A5A 0000	Test completed - OK
$5A5A FFFF	Test aborted - failure

6 Firmware Description: OK1-29K

All N.A.T. network boards are delivered with the multi-tasking kernel OK1 (Open Kernel 1). OK1 supplies all of the operating system resources required by the network software. In detail, it handles the:

- memory management
- time management
- signal handling
- task scheduling
- interrupt dispatching

One of OK1´s most important characteristics is its extremely short task switching times. This is a prerequisite for high speed handling of network protocols. OK1 is currently available for CPU cards based on the Motorola 680xy and AMD 29K.

6.1 Power Up Tests

The firmware tests all onboard components whenever the power is cycled or the reset switch is depressed. The test results are written to the “nif” field at the address = board´s base address + $4000.

The VMEbus line, SYSFAIL, will be driven accordingly: while the board is doing a reset, SYSFAIL will be active; after the reset and self-tests are completed, SYSFAIL will be inactive unless a failure has been detected.

The individual cells of the **“nif”** fields are assigned as follows:

Cell	Assignment
$4000	Status
$4004	Test Phase
$4008	Base Address RAM1
$400C	Maximum Address RAM1
$4010	Base Address RAM2
$4014	Maximum Address RAM2

The ***status*** cells can take the following values:

Value	Meaning
$5A5A 1234	Test in progress
$5A5A 0000	Test completed - OK
$5A5A FFFF	Test aborted - failure

If an error occurs during the tests, the number associated with the test phase that failed will be written in cell $4004. The complete test suit consists of the following phases:

Number	Phase
1	RAM1 Test
2	RAM2 Test
3	RAM3 Test (not present on the Eth29)
4	Test Ethernet controller and I/O
5	Test Ethernet controller DMA and Interrupt
6	End

If an error is detected during the RAM tests, the base address of the defective RAM bank will be written in address $4008 and the address of the defective location will be written in address $400C.
The complete test suite executes in 2 seconds or less. After the tests are completed, the status cell should contain one of the three valid status codes (see above). If this is not the case, then either the board has not properly started (supply voltages too low?) or there is a problem in the multiported communications RAM1.

6.2 OK1 Processes

An OK1 process can be any application program that has been written in “C” and compiled with the appropriate compiler (for the 29k processor : AMD29000). The header-files, library, and other aids, which are required to compile an OK1 process, are available upon request from N.A.T. GmbH. All N.A.T. network protocols are OK1 programs.

6.3 Process Scheduling

The kernel can serve any number of simultaneous processes (limited only by the amount of memory). A new process can be loaded and started at any time from the VMEbus via the multiported RAM. Each OK1 process can create any number of sub-processes (Threads), which will run as independent processes in the same process environment as the calling process.
Processes are not interruptable (with the exception of interrupt routines) and remain active until they suspend (Sleep) or terminate themselves.

6.4 Inter-Process Communications

Processes can communicate with each other using signals. If a signal is sent to a process which is in a “Sleep” state, the process will be awakened (reactivated). If a signal is sent to a

If an error occurs during the tests, the number associated with the test phase that failed will be written in cell $4004. The complete test suit consists of the following phases:

Number	Phase
1	RAM1 Test
2	RAM2 Test
3	RAM3 Test (not present on the Eth29)
4	Test Ethernet controller and I/O
5	Test Ethernet controller DMA and Interrupt
6	End

If an error is detected during the RAM tests, the base address of the defective RAM bank will be written in address $4008 and the address of the defective location will be written in address $400C.
The complete test suite executes in 2 seconds or less. After the tests are completed, the status cell should contain one of the three valid status codes (see above). If this is not the case, then either the board has not properly started (supply voltages too low?) or there is a problem in the multiported communications RAM1.

6.2 OK1 Processes

An OK1 process can be any application program that has been written in “C” and compiled with the appropriate compiler (for the 29k processor : AMD29000). The header-files, library, and other aids, which are required to compile an OK1 process, are available upon request from N.A.T. GmbH. All N.A.T. network protocols are OK1 programs.

6.3 Process Scheduling

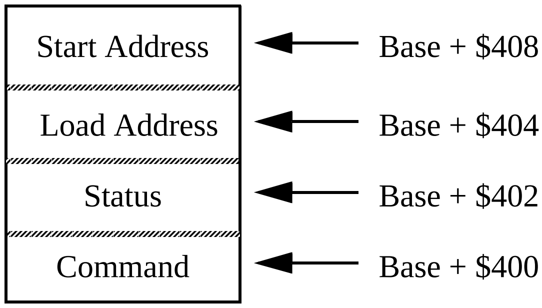
The kernel can serve any number of simultaneous processes (limited only by the amount of memory). A new process can be loaded and started at any time from the VMEbus via the multiported RAM. Each OK1 process can create any number of sub-processes (Threads), which will run as independent processes in the same process environment as the calling process.
Processes are not interruptable (with the exception of interrupt routines) and remain active until they suspend (Sleep) or terminate themselves.

6.4 Inter-Process Communications

Processes can communicate with each other using signals. If a signal is sent to a process which is in a “Sleep” state, the process will be awakened (reactivated). If a signal is sent to a process which is active (e.g. from an interrupt routine), the process will be reactivated immediately as soon as the sleep command is processed. Signals are <u>not</u> stored in a signal queue.

6.5 Starting an OK1 Process

A new OK1 process can be started using the appropriate parameter field in the multiported RAM of the slave board. The program code for the new process may already be present on the board or it must be loaded into the multiported RAM before being started. If the load and start addresses are not identical, the onboard CPU will first copy the code to the start address.



OK1 Parameter Fields

After the other cells have been written, the command cell can be written to start or stop the process. Once a process has been started, the slave board will write a zero in the command cell and a status code in the status cell.

Command Codes

Code	Command
1	start new task
2	stop task

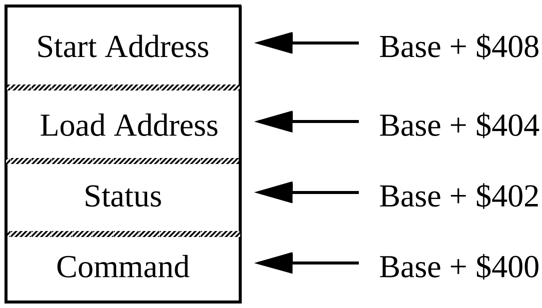
Status Return Codes

Code	Meaning
-1	task table full (only OK1-68)
-2	IRQ table full
-3	initialization error
-4	memory full
>=0	task number for the started process

process which is active (e.g. from an interrupt routine), the process will be reactivated immediately as soon as the sleep command is processed. Signals are <u>not</u> stored in a signal queue.

6.5 Starting an OK1 Process

A new OK1 process can be started using the appropriate parameter field in the multiported RAM of the slave board. The program code for the new process may already be present on the board or it must be loaded into the multiported RAM before being started. If the load and start addresses are not identical, the onboard CPU will first copy the code to the start address.



OK1 Parameter Fields

After the other cells have been written, the command cell can be written to start or stop the process. Once a process has been started, the slave board will write a zero in the command cell and a status code in the status cell.

Command Codes

Code	Command
1	start new task
2	stop task

Status Return Codes

Code	Meaning
-1	task table full (only OK1-68)
-2	IRQ table full
-3	initialization error
-4	memory full
>=0	task number for the started process

6.6 OK1 System Calls - OK1lib

The following system calls are provided in the form of a trap library, OK1lib, that is available for all application programs. All system calls are executed at a higher priority by the kernel and can not be interrupted.

FUNCTION VOID *OK1_SrqMem(nbytes)
request memory from system
returns: address of the memory block allocated
0 on error
ULONG nbytes; number of bytes requested

FUNCTION VOID OK1_SrtMem(buffer, nbytes)
return memory to the system
UCHAR *buffer; address of the block to be returned
ULONG nbytes; number of bytes previously requested with the call *C_SrqMem

FUNCTION int OK1_Sleep(ticks)
suspend execution of current task
returns: rest-time in ticks, if awakened by a signal
ULONG ticks; number of ticks (system dependent) to sleep

FUNCTION int OK1_Signal(Pid, Signal)
send a signal to the destination task
returns: -1 on error
USHORT Pid; process ID of the destination task
USHORT Signal; signal code

FUNCTION int OK1_AddSrv(SrvNum, SrvAdr)
insert a new service routine into the kernel´s service table
returns: -1 on error
ULONG SrvNum; service number of the new call
FUNCTION *SrvAdr(); entry address for the new service routine

FUNCTION int OK1_AddIRQ(vector, addr)
insert a new IRQ routine into the IRQ table
The users “C” language IRQ routine will have the same process environment as the calling process
returns: -1 on error
UCHAR vector; service number of the new call
FUNCTION *addr(); entry address for the new service routine

6.6 OK1 System Calls - OK1lib

The following system calls are provided in the form of a trap library, OK1lib, that is available for all application programs. All system calls are executed at a higher priority by the kernel and can not be interrupted.

FUNCTION VOID *OK1_SrqMem(nbytes)
request memory from system
returns: address of the memory block allocated
0 on error
ULONG nbytes; number of bytes requested

FUNCTION VOID OK1_SrtMem(buffer, nbytes)
return memory to the system
UCHAR *buffer; address of the block to be returned
ULONG nbytes; number of bytes previously requested with the call *C_SrqMem

FUNCTION int OK1_Sleep(ticks)
suspend execution of current task
returns: rest-time in ticks, if awakened by a signal
ULONG ticks; number of ticks (system dependent) to sleep

FUNCTION int OK1_Signal(Pid, Signal)
send a signal to the destination task
returns: -1 on error
USHORT Pid; process ID of the destination task
USHORT Signal; signal code

FUNCTION int OK1_AddSrv(SrvNum, SrvAdr)
insert a new service routine into the kernel´s service table
returns: -1 on error
ULONG SrvNum; service number of the new call
FUNCTION *SrvAdr(); entry address for the new service routine

FUNCTION int OK1_AddIRQ(vector, addr)
insert a new IRQ routine into the IRQ table
The users “C” language IRQ routine will have the same process environment as the calling process
returns: -1 on error
UCHAR vector; service number of the new call
FUNCTION *addr(); entry address for the new service routine

FUNCTION int OK1_GetBoardID(ethid)

get the onboard 6 byte Ethernet ID

returns: -1 on error

UCHAR *ethid; address where the Ethernet ID is stored

FUNCTION PD *OK1_GetPD()

get the address of own Process Descriptor

returns: address of process descriptor
-1 on error

FUNCTION void OK1_Exit()

terminates the task currently running and returns all allocated resources to the system

FUNCTION int OK1_Fork(addr)

Fork a private sub-process of the currently running process

returns: ID of the created process
-1 on error

void *(addr); entry address of the new sub-process

FUNCTION UCHAR *OK1_rmque(in, out)

remove one entry from END of FIFO queue

returns: address of the removed element
0 if there is no entry in the queue

UCHAR **in; address of the FIFO-IN pointer
UCHAR **out; address of the FIFO-OUT pointer

6.6.1 mbuf Handling Functions

FUNCTION char*
m_clalloc(ncl, how, canwait) **Allocate mbuf cluster space**
register int ncl;
int how;

FUNCTION int
m_expand(canwait) **Expand mbuf cluster space**
int canwait;

FUNCTION struct mbuf*
m_get(canwait, type) **Get an mbuf**
int canwait, type;

FUNCTION int OK1_GetBoardID(ethid)
get the onboard 6 byte Ethernet ID
returns: -1 on error
UCHAR *ethid; address where the Ethernet ID is stored

FUNCTION PD *OK1_GetPD()
get the address of own Process Descriptor
returns: address of process descriptor
-1 on error

FUNCTION void OK1_Exit()
terminates the task currently running and returns all allocated resources to the system

FUNCTION int OK1_Fork(addr)
Fork a private sub-process of the currently running process
returns: ID of the created process
-1 on error
void *(addr); entry address of the new sub-process

FUNCTION UCHAR *OK1_rmque(in, out)
remove one entry from END of FIFO queue
returns: address of the removed element
0 if there is no entry in the queue
UCHAR **in; address of the FIFO-IN pointer
UCHAR **out; address of the FIFO-OUT pointer

6.6.1 mbuf Handling Functions

FUNCTION char*
m_clalloc(ncl, how, canwait) **Allocate mbuf cluster space**
register int ncl;
int how;

FUNCTION int
m_expand(canwait) **Expand mbuf cluster space**
int canwait;

FUNCTION struct mbuf*
m_get(canwait, type) **Get an mbuf**
int canwait, type;

FUNCTION struct mbuf*
m_getclr(canwait, type)
int canwait, type;

Get a zeroed mbuf

FUNCTION struct mbuf*
m_free(m)
struct mbuf *m;

Free an mbuf

FUNCTION struct mbuf*
m_more(canwait, type)
int canwait, type;

Get more memory for mbufs

FUNCTION void
m_freem(m)
register struct mbuf *m;

Free multiple mbufs

FUNCTION void
m_clget(m)
register struct mbuf *m;

Get a page for an mbuf

FUNCTION char*
m_getpag()

Get a page without mbuf

FUNCTION void
m_putpag(m)
register struct mbuf *m;

Return a page without mbuf

FUNCTION struct mbuf*
m_copy(m, off, len)
register struct mbuf *m;
int off;
register int len;

Make a copy of an mbuf chain

FUNCTION void
m_cpydat(m, off, len, cp)

Copy data from an mbuf chain starting “off” bytes from the beginning, continuing for “len” bytes, into the indicated buffer.

register struct mbuf*m;
register int off;
register int len;
char *cp;

FUNCTION struct mbuf*
m_getclr(canwait, type) — **Get a zeroed mbuf**
int canwait, type;

FUNCTION struct mbuf*
m_free(m) — **Free an mbuf**
struct mbuf *m;

FUNCTION struct mbuf*
m_more(canwait, type) — **Get more memory for mbufs**
int canwait, type;

FUNCTION void
m_freem(m) — **Free multiple mbufs**
register struct mbuf *m;

FUNCTION void
m_clget(m) — **Get a page for an mbuf**
register struct mbuf *m;

FUNCTION char*
m_getpag() — **Get a page without mbuf**

FUNCTION void
m_putpag(m) — **Return a page without mbuf**
register struct mbuf *m;

FUNCTION struct mbuf*
m_copy(m, off, len) — **Make a copy of an mbuf chain**
register struct mbuf *m;
int off;
register int len;

FUNCTION void
m_cpydat(m, off, len, cp) — **Copy data from an mbuf chain** starting “off” bytes from the beginning, continuing for “len” bytes, into the indicated buffer.
register struct mbuf*m;
register int off;
register int len;
char   *cp;

FUNCTION void
m_cat(m, n)
register struct mbuf *m, *n;

Concatenate 2 mbufs

FUNCTION void
m_adj(mp, len)
struct mbuf *mp;
register int len;

Adjust an mbuf

FUNCTION struct mbuf*
m_pullup(n, len)
register struct mbuf *n;
int len;

Rearrange an mbuf chain, so that len bytes are contiguous and in the data area of an mbuf (so that mtod and dtom will work for a structure of size len).

FUNCTION void
m_cat(m, n)
register struct mbuf *m, *n;

Concatenate 2 mbufs

FUNCTION void
m_adj(mp, len)
struct mbuf *mp;
register int len;

Adjust an mbuf

FUNCTION struct mbuf*
m_pullup(n, len)
register struct mbuf *n;
int len;

Rearrange an mbuf chain, so that len bytes are contiguous and in the data area of an mbuf (so that mtod and dtom will work for a structure of size len).

7 Manual Revision History

RELEASE	DATE	SUBJECT
1.5	12.01.96	Added CAM support for FDDI29
		Added DMA support for FDDI29
1.6	20.02.96	Added direct FDDI chip set register access for CAM commands
1.7	04.06.96	Added Bits NOREL, NOMIN in L2_Init, dma_modes field

7 Manual Revision History

RELEASE	DATE	SUBJECT
1.5	12.01.96	Added CAM support for FDDI29 Added DMA support for FDDI29
1.6	20.02.96	Added direct FDDI chip set register access for CAM commands
1.7	04.06.96	Added Bits NOREL, NOMIN in L2_Init, dma_modes field