Magnet
A local network for Lilith computers

ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

Jiri Hoppe

MAGNET
A LOCAL NETWORK FOR LILITH COMPUTERS

November 1983

57

Address of the author:

Institut für Informatik
ETH-Zentrum
CH-8092 Zürich

MAGNET

A Local Network for Lilith Computers

Jiri Hoppe
Institute for Informatics
ETH Zurich

Abstract

The basic concepts and ideas of the local network *Magnet* connecting a number of personal computers *Lilith* are described. The rationals for the choice of the Ethernet system are discussed. The structure of the hardware design of the interface is shown. The three lower levels of the Magnet software are described and some details of the implementation are discussed. Several application programs are described.

Keywords:

local area networks, personal computers, communication, message exchange, remote files, remote procedure calls, Modula-2

1. Introduction

The introduction of the personal computer Lilith [Wir81] as a work-station at the Institut für Informatik of ETH Zurich changed significantly the working style of all users. A large amount of the work is now carried out locally on a personal station without any connection to the "outside world". The users of Lilith edit their programs, compile and execute them, type reports, draw schematics on their personal workstations without taking care of the activities of other users in neighbour offices. For some applications, however, communication among users is necessary: they work on a common project and need exchange of files containing the recent version of programs and data, they need access to the central file store, or they use some central devices like printers or tape stations.

The project **Magnet** was started to enable a communication between individual Lilith stations. The goal of the project was to provide a local network connecting up to hundred Liliths located in the main building of the Institute.

The network should connect mainly one type of computer Lilith, and eventually some other minicomputers. Connecting large mainframes, small microcomputers or even terminals is not considered. The network should be used for local communication only, there is no aim to provide a campus-wide network supporting a large number of different computers. The transmission speed of the network should be above 1 Megabit/sec in order to satisfy applications like remote disk access or picture animation. Since the network interface should be used in all Liliths, the hardware costs for each station should be kept low.

Section two of this paper describes the rationals to choose an Ethernet based network, the third section gives an overview of the hardware structure. The fourth section discusses the software; three levels of software are described. The last section decribes some application programs based on the Magnet network.

2. The Choice of the Network System

A local network connecting a small number of personal computers has already been implemented by several manufacturers. When the Magnet project was started, the possibilities of adopting an already existing system for Lilith were checked. The following systems were chosen for the closer evaluation: the Cambridge Ring[Hpr78], the Cobus System of Swiss Federal Institute of Technology Lausanne[Som76], the experimental Ethernet of Xerox [Met76] and our own development using commercially available HDLC chips.

There is a strong discussion in the literature about advantages and disadvantages of various systems for local communication. Numerous simulations prove that some systems are better than others. The author is of the opinion that for the expected applications of the network, and for the expected load, all above mentioned systems would give about the same performance. Any speed over 500kbit/sec would satisfy most applications, since at such speed the performance limitation is given by the slow software rather than by the actual transmission rate. A transmission of a packet of 128 words, each having 16 bits, takes about 2msec, assuming a transmission rate of 1 Mbit/sec. That is many times shorter than an average disk access and in about the same order as many software levels need to propagate the packet from or to the end user.

The argument that high speed is needed because the cable is used simultaneously by many users is hard to justify. Our measurements (see section 6) and experiences of other users of local networks show, that for the expected applications the average load of the system is very low. The cable is mostly used below 5%, thus keeping a lot of free capacity even for special applications.

It seems that differences of various methods will be noticed only by applications with strong real time restrictions, where the full speed of the transmission medium is needed. Since the Lilith computer is not used at the institute for any such applications, we do not expect problems where the transmission speed would be a severe limitation.

Last but not least, there are no aims to extend the network to become a universal network with thousands of stations over the whole campus. The network should stay strictly local.

At the end of the first part of the evaluation there were only two significant criteria for the choice of the system:

1) The analog part of the network (connection between the interface and the cable) should be simple and reliable. The transmission of high speed data over long distances is a complex problem needing a very careful design. Since there is only little knowhow at the institute about analog technology, we were anxious to adopt a commercialy available solution. Because of some unsolved analog problems the Cobus system was excluded from further evaluation.

2) The cost of the network should be kept low. The complete network interface should occupy only one board of the Lilith computer, this means it should consist of at most 70 chips. Since the version of the Cambridge ring available at that time definitely exceeded this complexity, it was excluded from further evaluation.

In summer 1980 we discovered that there was an "experimental" Ethernet interface being developed at Stanford University. This interface uses a very smart design by means of finite state machines; it requires only about 30 chips for packet handling including framing, CRC-check, and collision detection. To build a simple interface only two DMA channels are needed. Since the transcievers could be obtained off the shelf, thus solving the analog problem, it was decided to use the Ethernet for the local network of Lilith computers. The "experimental" Ethernet uses a transmission speed of 3 Mbit/sec. It is a predecessor of the 10 Mbit/sec Ethernet that is now standardized in [Eth82]. Both Ethernets base on the same principle, the signals on the cable and the packet formats are, however, incompatible.

3. The Hardware

An Ethernet network is based on the CSMA/CD (carrier sense multiple access with collision detection) principle [Met76]. All stations of the network are connected by a common serial bus. The access control is completely distributed, i.e. there is no central station giving permissions to access the bus. A station that is ready to transmit, listens to the signals on the cable. It waits until there is no traffic and immediately starts transmitting. Due to the distributed control, it is possible that more than one station is ready to transmit and therefore several stations start transmitting simultaneously, thus disturbing each other. Such a situation is called a

collision. In such a case all stations transmit an additional signal to *jam* the signals on the cable so that the collision will be certainly noticed by all stations and all stations will abort the transmission. Every station waits for a short time and repeats the transmission.

The hardware of the network system can logically be divided into the following parts: coaxial cable, transceiver, transmitter, receiver, and DMA channels to main memory with some control logic.

The coaxial cable connects all computers of a Magnet network. The length of the cable is limited to about 600 meters and there may be up to 100 computers connected to it. These limitations are given by the propagation speed of the signals (length) and by the reflexions caused by each transceiver (maximal number of stations). The cable is a 75 Ohm RG11 type as used for cable television.

The transceiver converts the analog signal from the coaxial cable to a digital signal with TTL level for the receiver or converts the TTL signal back to the analog signal to be transmitted on the cable. Additionally, the transceiver detects a collision - a state when more than one station tries to send on the cable.

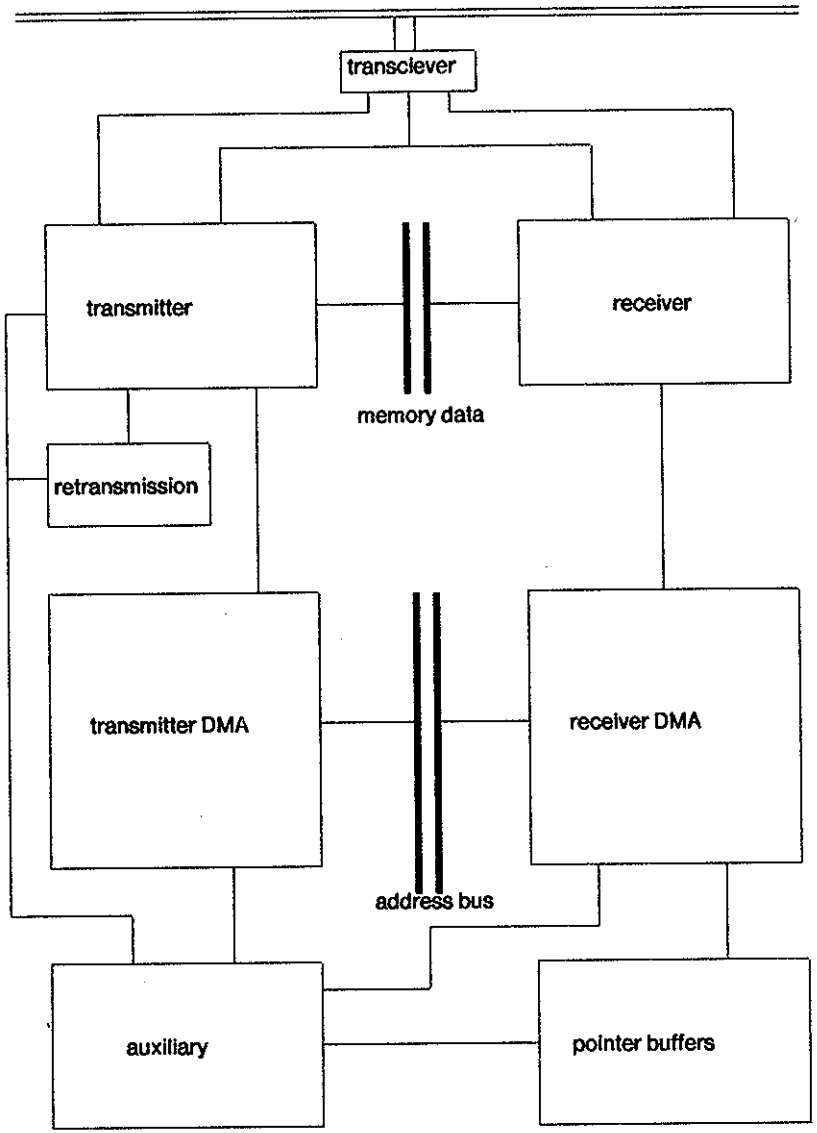
The transmitter listens to the traffic on the cable, and waits until no other station transmits. When the cable becomes free, the transmitter starts sending the data as they are supplied by the DMA controller. The CRC (cyclic redundancy check) is computed during the transmission and appended at the end of the packet. When a collision is detected during a transmission the signal on the cable is *jammed* , the transmitter abandons the packet, and the controller is notified about the collision.

The receiver listens to signals on the cable. When the start of a packet is detected, the receiver checks the first byte of the information containing the destination address of the packet. If this address matches the content of the address filter, the packet will be further processed, otherwise it is ignored. The incomming data are converted from the serial to the parallel form and passed to the receiver DMA controller. The CRC is computed and compared at the end of the packet with the CRC computed by the transmitter. If a difference is encountered, the packet is marked to have a parity error.

The data flow between main memory and transmitter/receiver part is controlled by two DMA channels, one for each direction. Both channels are built using the AMD 2940 chips containing all necessary counters.

The distributed control of the Ethernet system allows a situation where two or more packets addressed to one station may arrive very close together (*head-to-tail*) with a gap of only a few microseconds. Since the software cannot act that fast and prepare new buffers for the next packet, a hardware mechanism is built in allowing reception of up to 16 such *head-to-tail* packets without any software action. The packets are stored in a circular buffer in the main memory and the pointers to each buffer are stored in a first-in-first-out buffer. The software can later extract the packets from the circular buffer and reassemble them.

Upon notification of a collision the controller starts the retransmission algorithm. The controller waits a short period of time (in the range of about 50 microseconds) and tries to repeat the transmission. If another collision occurs, the controller waits twice as long and tries again. Up to seven retries are perfomed automatically by the hardware always waiting twice as long as the previous time. A station dependent



Hardware overview

Fig.1

generation of these time intervals assures that no two station wait in the same *phase*. If all seven tries were unsuccessfull, the hardware sends an interrupt and sets a bit in the status register, indicating the failure.

4. The Magnet Software

The Magnet software consists of a number of modules, enlarging the existing virtual machine of the Lilith's operating system **Medos-2** [Knu83] by introducing new features.

These modules may be divided into 3 hierarchical layers. They correspond to the lower 5 levels of the ISO (International Standard Organization) model. The level 6 (presentation level) is empty in the Magnet system since there is the same representation of data in the homogeneous environment of Lilith computers. Level 7 (application level) is not considered to be a part of Magnet. The Magnet system does not exactly follow the recommendations of ISO. The system is kept less general but tailored to the requirements of a local network for personal computers. In this way, a large amount of the overhead needed in a heterogeneous environment is omitted.

The lowest level of the Magnet software is built by modules handling the basic flow of information between processes. They describe the format of information and the flow of data between a common buffer pool, drivers, and user processes.

The next level provides procedures for locating of *named services* in the network and maintaining connections between such services.

The above two levels are used by system programs only and are not accessible to normal user programs. Such programs use a number of network access methods that are implemented in the third level. A user can choose the representation of the network, that matches in the best way his applications. Currently, three network access methods have been or are being implemented: remote messages, remote files, and remote procedure calls.

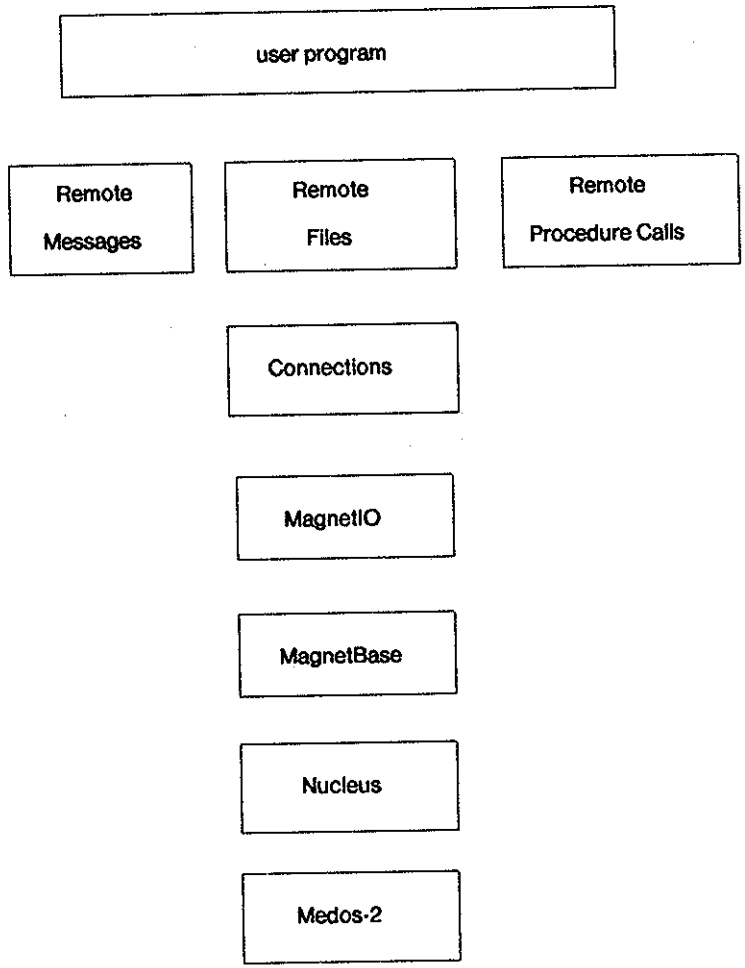
These three levels of Magnet are enlarged by a module *Nucleus* providing functions needed for multiprogramming. The module Nucleus can be considered to belong to the underlying operating system.

Note: The description of software objects in this report does not always exactly correspond to their declaration in definition modules. For reason of clarity same objects are not mentioned at all, for some other objects different names are used that can be better understood by ther reader, or some procedures have simplified parameter list.

4.1. Low Level Software

4.1.1. Module Nucleus

The module Nucleus builds the lowest level of the system. The current version of Lilith's operating system Medos-2 [Knu83] supports only a primitive handling of concurrency. A network system, however, requires a comfortable support of concurrent activities including data transfer between processes, time out handling, process scheduling, etc. To satisfy such needs, a Nucleus with operations



Software overview

Fig. 2

satisfying the network requirements was developed to enhance the functions of Medos-2.

The Nucleus represents the concurrent activities as processes sharing one processor. Some of these processes may be interrupt driven. A process is created by a call of procedure

```
PROCEDURE CreateProcess(processCode : PROC;
                        stackSize   : CARDINAL;
                        VAR done    : BOOLEAN)
```

The procedure *processCode* contains the code of the process, *stackSize* determines the size of the working stack of the process; *done* designates the result of the creation.

The processes communicate by passing messages through a common port. A process can send a message to a port by a call of the procedure

```
PROCEDURE SendMsg(VAR p          : Port;
                  VAR answerPort : Port;
                  VAR msg        : Message)
```

If a process is waiting for a message in a port *p*, it will be resumed by a call of *SendMsg*. If no processes are waiting, messages are stored in the port. The *answerPort* is used in *master-slave* relations to identify the port, where the answer from the slave back to the master should be sent to. The identification of the *answerPort* is transferred inside a message and can be extracted from the message by the call of the procedure

```
PROCEDURE GetAnswerPort(msg : Message): PortPointer
```

A process may wait for a message by

```
PROCEDURE WaitMsg(VAR p : Port; VAR msg : Message)
```

If there are no messages stored in the port, the process calling *WaitMsg* is delayed and resumed when the first message arrives. If there is at least one message in the port at the time of the call, the process is not delayed, it fetches the message and continues processing.

A similar procedure allows processes to wait for a message but to specify a maximal time limit to wait (timeout). If no message arrives within the specified time, the process is resumed and notified about the failure.

```
PROCEDURE WaitMsgTimeOut(VAR p       : Port;
                         VAR msg     : Message;
                             timeout : CARDINAL;
                         VAR ok      : BOOLEAN)
```

The *timeout* is given in multiples of 20 msec, *ok* specifies whether the procedure was terminated due to an arrival of a message or due to a time out.

The device drivers are represented as processes, each of them controlling one device. They are declared in a module with priority higher or equal to the interrupt priority of the device. Such processes may wait on a device interrupt by a call of

```
PROCEDURE WaitIO(dev : DeviceNumber)
```

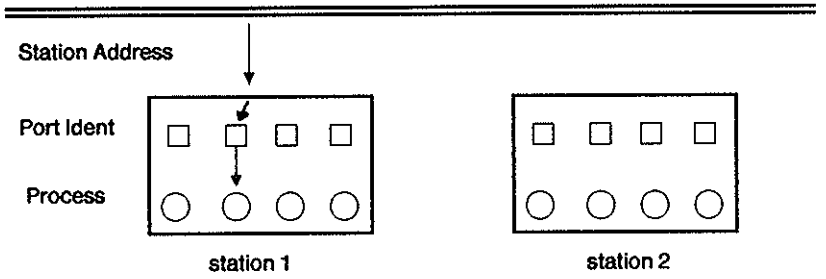
The parameter *dev* is an identification of a device. It may be interpreted as an interrupt vector.

4.1.2 Addressing

The network services are performed by service processes residing in different computers on the network. A network service is addressed by two fields in the header of packet: the *hardware address* of a station and an identification of a *port*. In the future a third field will be included: identification of a network when more Magnet networks will be interconnected.

The *hardware address* is a number between 0 and 255. A station may be opened to a combination of such addresses, but in most cases only two addresses are opened: the own address, as determined by the hardware identification of the interface (switch), and the broadcast address, accepted by all stations. The station address of the packet is interpreted by the hardware. Only packets with addresses matching the content of the hardware address filter are accepted for further processing .

The *port* is used to identify one of many services that are implemented within one station. In most cases there is only one process waiting for packets from one port, we may say that in this case the port is used to identify a process.



Addressing

Fig.3

There are two kinds of ports reserved for arriving packets.

1) *predefined ports* reserved for fixed services like name server, file system, remote procedure calls etc.

2) *private ports* created by user application programs.

Most network procedures handling with ports use the following identification of ports: The parameter designating the port is of type ADDRESS. If the value of this parameter is a small number (currently less than 16), it is assumed that it designates a *predefined port*. A larger value designates an address of a *private port*. In the following parts of the report we call the type used for port addressing *PortIdent*.

Ports are opened by a call of

```
PROCEDURE OpenMagnetPort( portId   : PortIdent;
                          VAR done : BOOLEAN)
```

When a port is no longer used it is closed by a call of

```
PROCEDURE CloseMagnetPort(portId   : PortIdent;
                          VAR done : BOOLEAN)
```

If a program terminates without closing its ports the system will close them automatically.

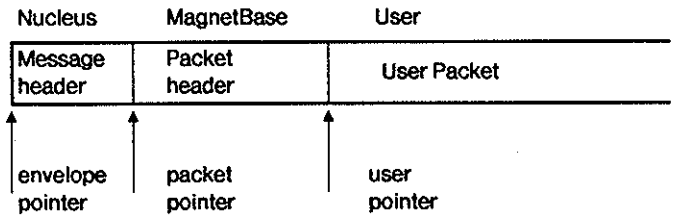
4.1.3 Basic Data Types

The main flow of information between local processes of the Magnet system is performed by messages containing communication packets. In order to hide a direct access to the structures of the Nucleus, the messages are renamed at the Magnet level to *envelopes*. It shows a similarity of the packet transport with postal services.

Envelopes are structured objects with different access rules depending on the level of the network system. There are three levels of access (see fig. 4): At the Nucleus level only the header of an envelope is defined, the other part of the envelope contains data with unknown structure. The next two levels have no access to the header of an envelope, since this is defined as a hidden type in the Nucleus. In these levels a structured access to the content of the envelope - to the packet - is provided. Such a packet consists of two parts: the header of a packet and a user specified part. The header is common to all magnet applications. It contains the destination and source addresses, sequence numbers, etc.

```
Packet =
  RECORD
    destAdr    : CARDINAL;  (* destination address *)
    srcAdr     : CARDINAL;  (* source address *)
    destPort   : PortIdent; (* destination port *)
    returnPort : PortIdent; (* reply port *)
    connKeys   : INTEGER;   (* idents for connections *)
    xsum       : BITSET;    (* soft checksum *)
    seqNr      : CARDINAL;  (* sequence number *)
    pLength    : CARDINAL;  (* length of the packet *)
    packetType : CARDINAL;  (* type of the packet *)
  END
```

The user part is left unspecified at the lower level of Magnet. Its content is defined separately for each application, mostly as a record type with variant parts. This part is appended by the system to the basic packet.



Structure of a packet

Fig.4

The basic idea of introducing three levels of access is to provide a clear structuring of the network by hiding low level structures from unauthorized access from higher levels of system. This method provides in all levels of system a structured access to all fields of a packet with high-level programming techniques. The Magnet system never uses an *assembly-like* access to a packet represented as an unstructured sequence of bytes. The structured definition increases the reliability of the system and enforces clear communication among different programs.

The language Modula offers the following method for such a hierarchical structuring of the data access: The access to the envelope and to both parts of the packet are performed by three pointers: *envelope-pointer, packet-pointer,* and a *user-packet-pointer*. The first two pointers are *type bound* to types defined in modules Nucleus resp. MagnetBase. The third type is left unspecified, it is defined as a universal type ADDRESS. Such a type is compatible with all pointer types in Modula-2. A user may therefore declare a pointer to his private type and assign this pointer to the user part of the packet.

Such a method offers good flexibility. It is, however, a potential source of serious problems. The user part of an envelope has a maximal size, defined at the time of the generation of the system. If the size of the user type fits into this space no problem occurs. If, however, the size of the user type exceeds the maximal size, the user may overwrite data outside of the envelope. To prevent such run time errors every procedure providing the user with envelopes checks the size of the user type.

4.1.4 Basic Flow of Information

There is a closed flow of envelopes among all Magnet processes maintained by a common pool of empty envelopes. A process needing an envelope for communication requests it by a call of

```
PROCEDURE GetEmptyEnvelope(VAR env      : Envelope;
                           VAR packet   : PacketPtr;
                           VAR userInfo : ADDRESS;
                               maxSize  : CARDINAL)
```

The *packet* and *userInfo* points to the packets encompassed in the *env*. After the use, the envelope is returned back to the pool by

```
PROCEDURE ReturnEmptyEnvelope(VAR env : Envelope)
```

Both procedures are implemented by means of a port *emptyPort* where envelopes are sent by *SendMsg* or requested by *WaitMsg*.

Such a common pool of envelopes has advantages for the network system, but unfortunately introduces also some problems. The advantage is that the memory is better utilized, since more processes share the common pool, and that there is an automatic way of buffering during the communication between processes. A number of incomming packets is stored in a port until the target process becomes ready and fetches them.

There are two basic problems: A common pool with a fixed size is a potential source of deadlocks when some processes are programmed in a way that they may posses more than one envelope. The other difficulty is a complicated handling of situations

where a process acquires an envelope and dies due to a run time error before the envelope is returned back. Such envelopes must be retrieved by the system that keeps information about the current owner of an envelope.

The experience gained shows that a common buffer pool is a better solution than a system where every process is responsible for its buffers. The above mentioned disadvantages are handled by careful design of processes and by routines retrieving lost envelopes.

4.1.5 The Module MagnetBase

The module *MagnetBase* declares the type *Packet*, the procedures handling the common envelope pool, and the administration of open ports as described above. Additionally the implementation body of the module *MagnetBase* includes the drivers for the receiver and the transmitter. Both drivers are implemented as concurrent processes. The communication among other processes of the network system and the drivers is performed by exchange of messages through communication ports. A process ready to transmit a packet sends an envelope containing the packet to the *xmitReq* port. The transmitter fetches the envelope from this port, transmits the enclosed packet, and returns a message to the source process with a status information indicating whether the transmission was successful or failed due to too many collisions.

The transmitter has the following structure:

```
PROCESS Transmitter;
  VAR env : Envelope;
      pp  : PortIdent;
  BEGIN
    LOOP
      (* wait for a packet to be sent *)
      WaitMsg(xmitReq, env);
      pp := GetAnswerPort(env);
      (* set DMA info into 2940 *)
      DMA.info := {ADR(packet); length(packet)};
      start transmission;
      WaitIO (xmit);
      env↑.reply := status of the tranmission;
      SendMsg(pp, env)
    END
  END Transmitter;
```

The structure of the receiver process is more complicated than the structure of the transmitter. The incoming packets are stored by the hardware in a circular buffer in the main memory and the pointers to each packet are stored, together with the status information, in a FIFO buffer. The receiver process takes this information from the buffer, and checks the status. If the packet was received correctly, it is copied into an envelope buffer and sent to a waiting process according to the destination port specified in the header of the packet.

The process *Receiver* has the following structure:

```
PROCESS Receiver;
  VAR
    env      : Envelope;
    packet   : PacketPointer;
    userInfo : ADDRESS;
  BEGIN
    LOOP
      WaitIO(rec);
      fetch all packet pointers from the FIFO;
        FOR all pointers DO
          IF (statusInfo = noErrors) AND
             (No overflow in circular buffer) THEN
            GetEmptyEnvelope(env, packet, userInfo);
            copy packet from the circular buffer
              to the packet;
            IF packet↑.destPort is opened THEN
              SendMsg(packet↑.destPort, packet);
            ELSE
              ReturnEmptyEnvelope(env, packet, userInfo);
            END
          END
        END
      END
    END Receiver;
```

4.1.6 Exchange of Packets - the Module MagnetIO

The module *MagnetIO* provides procedures for higher level communication with the two driver processes. It hides the structuring of the system into processes and the using of ports to communicate with drivers. The communication with these processes is established by the two procedures

```
PROCEDURE Transmit(VAR envelope : Envelope;
                   VAR reply    : Reply)

PROCEDURE Receive(VAR port     : Port;
                  VAR envelope : Envelope;
                  VAR packet   : PacketPtr;
                  VAR userInfo : ADDRESS;
                      timeout  : CARDINAL;
                      maxSize  : CARDINAL);
```

The procedure *Transmit* computes the soft-checksum, and sets the source address in the packet. All other fields must be already filled by the higher levels of the Magnet system. Next, the transmitter is activated by sending the envelope to the *xmitReq* port.

The procedure *Receive* waits for an *envelope* from the receiver process, addressed to a *port*, and makes some low level checking, like checksumming or ignoring damaged packets. The *packet* points to the packet information enclosed in the envelope and *userInfo* points to the user part of the packet. The *timeout* specifies

the maximal time to wait for a packet and *maxSize* is used to check the maximal size of the user part of the packet (see 4.1.3).

A simple protocol for an often used information exchange, where a master sends one request that must be acknowledged by exactly one answer from a slave, is handled by the procedure

```
PROCEDURE TransmitAndRecieve
          (    commDesc : CommunicationDescriptor;
           VAR xmitEnv  : Envelope;
           VAR recEnv   : Envelope;
           VAR packet   : PacketPtr,
           VAR userInfo : ADDRESS;
               maxSize  : CARDINAL)
```

The *commDesc* keeps some constants for the protocol (timeout length, sequence number, receiver port, etc.), *xmitEnv* encompasses the request packet to be transmitted, *recEnv* contains (possibly after some retries) the received packet. The procedure *TransmitAndRecieve* sends a packet contained in *xmitEnv* to the slave station and waits for an answer. If a packet arrives with a sequence number corresponding to values specified in *commDesc* it is copied to the *recEnv* and the procedure is left. If no packet is received during a period of time given by *timeout* or a wrong packet arrives, another *xmitEnv* is sent and the procedure waits again for an answer. This cycle is repeated until either a correct packet is received or the maximal number of retries is exceeded.

4.2 Connections

Only very few applications of Magnet exchange single, individual packets with no relation between consecutive packets. Mostly a large number of related packets, belonging to the same session, is exchanged in a row. We call such a session a *connection*.

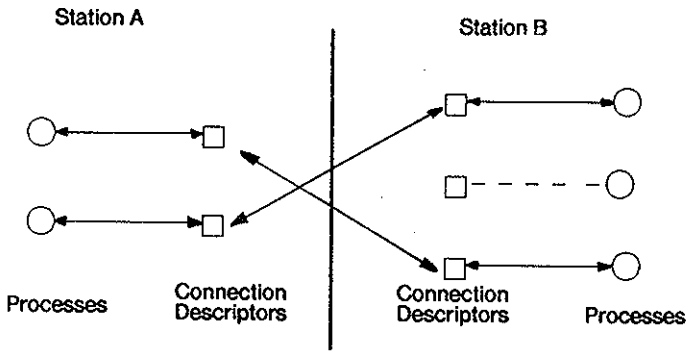


Fig.5

A connection establishes a communication channel between two processes by

exchange of opening packets. The actual communication on this channel is carried out by means of similar procedures as defined in the Module *MagnetIO*. The important difference is that the header of the packet is now hidden to the user and it is copied by the module *Connections* from a connection descriptor. At the end of a session the communication channel is closed.

An important feature of a connection is the possibility to check the status of the partner station any time to find out if it is still active, or inactive due to hardware problems, abortion, etc.

The module *Connections* contains an array of *connection descriptors*. They are allocated when a connection is opened. For the user, a connection is represented by a *connection identifier* which is actually an index to the above array.

Connections are established between *named services* only. This means that every service must be known by its name in the network before it is accessed. The module *Connections* provides facilities to locate such services and to return their *characterization* consisting of the hardware address and the identification of the port (see 4.1.2.). A service is installed in the network by the call of

```
PROCEDURE InstallService(     name        : ARRAY OF CHAR;
                              portId      : PortIdent;
                          VAR serviceInfo : ARRAY OF CHAR;
                          VAR result      : Result)
```

The *name* designates the logical name of the service, *portId* is the identification of the port (see 4.1.2) and *serviceInfo* is text information used differently by various services. This procedure sets the *name* into a small memory resident data base and checks if there is no other station in the network with the same name. In case there is already such a name known in the network, the service is removed, since there cannot be two services with the same name.

A connection to an already installed service is opened by the call of

```
PROCEDURE OpenConnection(     partnerName : ARRAY OF CHAR;
                              localPortId : PortIdent;
                          VAR connId      : ConnectionId;
                          VAR serviceInfo : ARRAY OF CHAR;
                          VAR result      : Result)
```

The *partnerName* is the name of the partner service, *localPortId* is the identification of the local port. The *connId* is a connection identifier that is later used for packet exchange. This procedure locates first the named service and gets its characterization. Next, packets are exchanged that open connections and allocate descriptors in both stations. The descriptors are initiated with values describing the characterization of the partner station.

The location of the named service is based on the following principle: The procedure sends a packet with a broadcast destination address of the *name server* type containing the searched name. This packet is received by all active stations of one network and dispatched there to a process handling name requests. This process compares the name specified in the request with the internal data base containing the logical names of local services. If a match is found, the process sends a packet back to the sender with his physical address and the corresponding

port identification. If no match is found no reply is sent back. If no packet is received within a certain time interval, the sender retries several times. If several retries failed, it is assumed that there is no active service with the given name.

The state of a connection can be tested by

```
PROCEDURE ConnectionActive(connId : ConnectionId) : BOOLEAN
```

The state of connections is internally monitored by a process sending test packets in regular intervals to the partner station. If the partner station is active and the corresponding connection is still opened the test packet is acknowledged. If the test packet does not acknowledge several times the process assumes that the connection was aborted and marks the local descriptor as inactive. This mark is read by the procedure *ConnectionActive*.

Above the level of the module *Connections*, the users have no access to the header of a packet (see 4.1.3). They only access the *user part* of a packet. All procedures of the module *Connections* use therefore only a *user pointer* instead of an *envelope pointer*.

The flow of empty envelopes is maintained by the procedures

```
PROCEDURE GetEmptyPacket(VAR userPtr : ADDRESS;
                             maxSize : CARDINAL)
PROCEDURE ReturnEmptyPacket(VAR userPtr : ADDRESS)
```

They correspond to the similar procedures of the module *MagnetBase*.

The exchange of packets between stations is performed by procedures

```
PROCEDURE Transmit(    connId   : ConnectionId;
                   VAR userPtr  : ADDRESS;
                   VAR packetPar: PacketParam)

PROCEDURE Receive( VAR port     : Port;
                   VAR connId   : ConnectionId;
                   VAR userPtr  : ADDRESS;
                   VAR packetPar: PacketParam;
                       maxSize  : CARDINAL)

PROCEDURE TransmitAndReceive
                  (    connId      : ConnectionId;
                   VAR xmitUserPtr : ADDRESS;
                   VAR recUserPtr  : ADDRESS;
                   VAR packetPar   : PacketParam;
                       maxSize     : CARDINAL)
```

These procedures correspond to the equally named procedures in the module *MagnetIO*. The *connId* describes the connection, *packetPar* is used to keep a copy of some information that is contained in the now inaccessible header of a packet (length, sequence numbers, hardware status of a received packet, etc.)

4.3. Basic Communication Methods

The described modules build a basic structure used by all communications in the network. The normal users of the network are, however, not encouraged to access these levels directly. They should use the network via three basic network access methods: *remote messages*, *remote files*, and *remote procedure calls*.

4.4. Remote Messages

Remote messages offer to the user a similar service of exchanging data as offered by the module Nucleus. In Nucleus, processes communicate by exchanging messages through ports. Such processes and ports must be declared in the same processor. The module *RemoteMessages* enhances this communication by a possibility of message exchange between different processors.

There is no higher protocol implemented at this level, the messages are delivered according to the *best effort strategy*. The system tries its best to deliver a message but does not guarantee a secure arrival. This must be achieved by user implemented higher protocols. The module *RemoteMessages* operates only as a transport service for data beween two stations. Such a free exchange of messages allows the user to experiment with Magnet, to introduce new concepts, or to implement new protocols.

The structure of the module *RemoteMessages* is very similar to the structure of the module *Connections*. The main difference is another approach to the handling of data contained in a packet. Up to the module *Connections* the data are enclosed in envelopes that are required before and returned after use. In this way it was possible to transfer data through many levels of the system and through many processes without copying. At the level of *RemoteMessages* the data are handled as a universal type ARRAY OF WORD with unspecified content. The user supplies his private variables for transmission or reception and the module copies the data to or from the envelope.

The module *RemoteMessages* has a similar representation of connections as the module *Connections*. Two kinds of connections are supported: The *one-to-one* connection is used between two equal stations (e.g. two stations playing a game of chess remotely). The *N-to-one* connection is used for *master-slave* applications where one slave services a number of users.

A connection between two stations is opened by the call of

```
PROCEDURE OpenConnection
        (srcName, destName : ARRAY OF CHAR;
         VAR connDesc      : ConnectionDescriptor;
         timeOut           : CARDINAL)
```

The *srcName* and *destName* specify the own and the remote station names. These names are temporary inserted as service names into the network and after a successful opening removed again. The *connDesc* is used to keep all information necessary for message exchange (e.g. station address, remote port identification) and a port that is used to receive packets. The opening procedure includes a protocol with retransmissions and timeouts.

When a connection has been established the communication is carried out by calls

of two procedures

```
PROCEDURE SendRemoteMsg
          ( VAR connDesc : ConnectionDescriptor;
                data     : ARRAY OF WORD;
                length   : CARDINAL)
```

and

```
PROCEDURE WaitRemoteMsg
          ( VAR connDesc : ConnectionDescriptor;
            VAR data     : ARRAY OF WORD;
            VAR length   : CARDINAL;
                timeout  : CARDINAL)
```

Their semantic corresponds to the similar procedures of the module Nucleus. The *SendRemoteMsg* sends the *data* to the remote station specified in *connDesc*. The *WaitRemoteMsg* waits until a packet arrives from the station specified in *connDesc* and copies its content into *data*. The *timeout* specifies the maximum waiting time in multiples of 20 msec. The data may have an arbitary length. In case the data do not fit into a single packet, they will be transfered in more packets.

The *N-to-one* connection is used for *master-slave* applications. A typical example of this kind is a data base server that accepts queries from a number of users, processes these queries, and sends replies back to the user. Such an application must handle a large number of connections concurrently and must be able to open and close new connections every time. Additionally to the connection descriptors used for the network communication, the slave has a local descriptor for each connection. This descriptor is used to keep private data like pointers to the opened files for data base access. To allow the identification of local descriptors, every message contains an index of the local descriptor.

The remote message system provides for *N-to-one* connections the following method: the user uses for the communication the same remote message protocol as described above. The server calls the procedure

```
PROCEDURE Listen(name          : ARRAY OF CHAR;
            VAR userConnDescr : ARRAY OF
                                  ConnectionDescriptor;
                workProc      : WorkProcedure;
            VAR nameOk        : BOOLEAN)
```

This procedure is a kind of combination of *OpenConnection* and *WaitRemoteMsg*. This procedure installs itself as a service *name* in the system. The array of *connectionDescriptors* is used to keep information about every connection. The body of the procedure contains a semi-infinite loop. In this loop the slave waits for any arriving packet. If this packet is of the *open connection* type, a new connection descriptor from the field *userConnDescr* is allocated, and a confirmation packet is sent back to the user. If a packet with data arrives, the procedure finds the corresponding descriptor and calls a procedure *workProc*. This procedure has a type

```
TYPE WorkProcedure =
     PROCEDURE(ConnectionDescriptor, VAR ARRAY OF WORD,
               VAR CARDINAL)
```


The first parameter specifies the communication descriptor, the second where the arriving data should be copied into, and the third is the identification of the connection used to find the local descriptor. The server now processes the request and sends back the answer to the user identified by the connection descriptor using *SendRemoteMsg*.

4.5 Remote Files

The *remote files* method supports access to files residing in a remote station in a way identical to how files on a local medium (disk) are accessed. In order to understand this method it is necessary to discuss the operation principle of the general file system of the Medos-2 operating system.

4.5.1 The General File System of the Operating System Medos

The file system of the operating system Medos-2 was designed to support a number of various devices in such a way that a uniform interface to all drivers from user programs is achieved. The file system enables installation or removal of device drivers dynamically during the execution of a program, without the need for a time consuming system reconfiguration or a rebooting of the system. Typical devices are disks, magnetic tapes, or printers.

A Medos file is represented in a user program by a record variable containing pointers to the data buffers, information needed by the file system to connect a file with its device driver, and an area for parameter exchange between the user program and the driver. The data buffers are owned and maintained by each driver in order to enable efficient buffer handling even for more complicated applications (e.g. read ahead):

A file descriptor has the following structure:

```
TYPE File =
  RECORD
    (* buffer pointers *)
    bufAdr, elemAdr, inAdr, topAdr : ADDRESS;
    (* communication parameters between user and driver *)
    eof      : BOOLEAN;
    result   : Response;
    CASE com : Command OF
      create, open :
        fileNr, versionNr : CARDINAL;
    | lookup, rename :
        new : BOOLEAN
    | setRead, setWrite, setModify:
    | setPosition, getPosition:
        pos : Position
      ....
    END;
    (* connection between file and driver *)
    mt        : MediumType;
    subMedium : ADDRESS
  END
```

The operations on files are performed through a collection of procedures declared in the module *FileSystem*. These procedures may be divided according to their function into the following groups:

- general file handling: allows creation, deletion, renaming, setting, and reading of the current position, setting of the working mode (read, write, modify), and performing of a block data transfer according to the working mode.

- stream operations: perform sequential reading and writing of data of either word or byte size.

- direct call procedures enable direct activation of a device driver. This feature will be explained later in this text.

- installation and removal of drivers.

4.5.1.1. Information Flow Between User and Driver

The exchange of information between a user program and different device drivers is performed according to the following principle:

Every device driver is installed in the file system by submitting three entities :

- the device identification consisting of two letters, optionally followed by a number (e.g. 'DK' or 'MT1')

- two procedures which will be activated by the file system when actions by a specific driver are required. They have the following type:

```
TYPE
  FileProcedure      = PROCEDURE (VAR File);
  DirectoryProcedure = PROCEDURE (VAR File, ARRAY OF CHAR);
```

The *DirectoryProcedure* is called with a string parameter specifying the name of a file. Such a name is needed in case of opening or renaming of an existing file. All other activities are performed by the *FileProcedure*.

The names of files are structured in a way similar to qualified identifiers in the language Modula-2. They consist of a sequence of simple identifiers separated by dots. The first identifier of a file name specifies the device.

Example:

```
DK.temp.MOD
    is the file "temp.MOD" on a device "DK"

MT1.john.mary.ann.joseph
    is the file "john.mary.ann.joseph" on the device "MT1"
```

Only the device name is interpreted by the file system, the other part of a file name is left for interpretation by each driver.

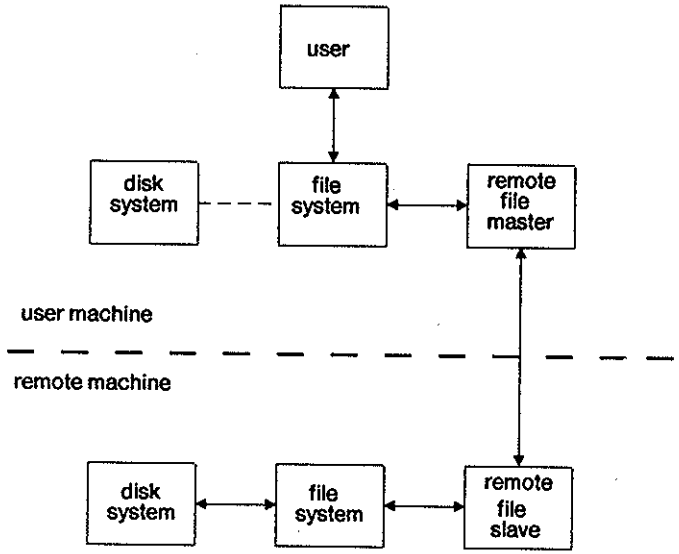
When a user calls a *general* (see above) procedure of the file system, the parameters of his call are copied to the *parameter* fields of the file descriptor, the appropriate driver is chosen and its *FileProcedure* or *DirectoryProcedure* is called. This procedure performs the action, transfers data (if necessary) to or from a buffer, and writes the result into the *result* field in the file descriptor.

The *direct call* procedures select the appropriate driver and activate its *FileProcedure* or *DirectoryProcedure*. It is assumed that the user already has copied

all parameters into the *parameter* fields of the file.

4.5.1.2. Access to Remote Files Through the General File System

The access to remote files is performed in the same way as the access to any other device. The general file system has no notion of the network, it accepts only an installation of the Magnet driver and forwards all requests for remote files to this driver. The user of a remote file sees only one difference between a local and a remote file: the device specification in the file name.



Data path during the remote file access

Fig.6

4.5.1.3. Naming of Remote Files

The magnet network system provides procedures for installation of a remote file server as a device in the general file system. As mentioned in the description of the file name, the first identifier in the composed file name specifies the device. This identifier is only two characters long. Since two characters are too short for a unique neetwork identification of a remote server, a concept of *nicknames* was introduced. A nickname is a free chosen two letter identification which is used as an abbreviation for a full service name as described in 4.2. A nickname is used as a device specification in a file name, but the remote file driver keeps the full service name for the procedure *OpenConnection* (see 4.2)
A remote file server identifies itself by making its name known to the network. For example such a server may be called *John* to identify the disk cartridge of John, or *centralFS* to identify the central file store. A user of remote files connects the logical name of a file server (e.g. John or centralFS) to a freely chosen device name (two characters: e.q. AB or XY), and informs the local file system, that all requests

concerning the chosen device name (e.g. AB or XY) should be forwarded to the remote file driver.
A connection between a nickname and a full name of a remote file system is perfomed by a call of

```
PROCEDURE InstallRemoteMedium(nickName   : ARRAY OF CHAR;
                              remoteName : ARRAY OF CHAR;
                              VAR done   : BOOLEAN)
```

This procedure builds a local table keeping both the nickname and the name of the remote file system and informs the file system about an installation of a new medium *nickname*

4.5.2. Implementation of Remote Files

The file exchange is performed by two modules: the *FileMaster* performing operations in the users computer, and the *FileSlave* performing operations in the remote computer (see Fig. 6).

The *FileMaster* implements two procedures *FileCommand* and *DirectoryCommand* which are submitted to the general file system for access to remote files. All nicknames share the same two procedures. When activated, these two procedures check the integrity of a request, maintain local descriptors for each opened file, and transform parameters of procedure calls to packets that are transmitted to the partner station. These packets may optionally contain data.

The structure of a packet for file services is the following:

```
FilePacket =
  RECORD
    slaveId  : FileId;  (* id of the remote file *)
    masterId : FileId;  (* id of the master file *)
    fcEof    : BOOLEAN; (* end of file *)
    fRes     : Response;
    CASE PacketType OF
      fileIdentification :
          CASE Command OF
            open, create:
              fileNr, versionNr : CARDINAL;
              (* description of the remote medium *)
              mediumType        : MediumType;
              mediumNr          : CARDINAL;
          | lookup, rename:
              fileName : MagFileName;
          END;
    | fileOperation:
          CASE Command  OF
            setpos, getpos, length :
              highPos, lowPos : CARDINAL;
          | ....
          | ....
           END;
          inCount : CARDINAL; (* # of bytes read in *)
```

```
      elCount : CARDINAL; (* current # of bytes *)
      CASE withData   : BOOLEAN OF (* data transport *)
        TRUE : fileData : ARRAY [0..maxData-1] OF WORD
      | FALSE:  (* nothing *)
      END;
  END; (* CASE packetType *)
END;
```

The packets for remote files may be of two types. The *fileIdentification* packets are used for actions where the identification of a file is necessary (open, lookup, rename,..). The *fileOperation* packets are used for all other actions.

The packets are received on the other station by a process *FileSlave*. This process checks the consistency of incoming packets, maintains local descriptors for every opened file, and converts received packets to calls of the appropriate local device driver. When the driver action has been finished, a reply packet is sent back to the master. This packet may optionally contain data.

The path of a file request through the system is shown in the following example:

A user on the station 'A' wants to copy files from a disk cartridge called *John.* He activates the name server that locates this cartridge at station 'B' and locally installs a freely chosen medium 'XY' to represent the cartridge of John. Now he looks up the file *temp.MOD* on the remote cartridge. The file name on the local station has the name *XY.temp.MOD*. On the remote station this file name will be converted to *DK.temp.MOD* in order to get to the disk (DK) driver.

The information enclosed by {...} in this example is an informal description of some kind of parameter transmission.

actor	action
A.user	Lookup(file, 'XY.temp.MOD', FALSE)
A.FileSystem	DirectoryCommand {magnetdriver,lookup,'XY.temp.MOD'}
A.RemoteFileMaster	convert name to DK.temp.MOD, find station with nickname XY => B open connection to B
A.RemoteFileMaster	SendPacket {connection=B, fileIdent, lookup, 'DK.temp.MOD'}
A+B.MagnetDrivers	A.transmit and B.receive packet
B.RemoteFileSlave	receive packet, check parameters
B.RemoteFileSlave	DirectoryCommand { DK, lookup, 'temp.MOD' }
B.DiskSystem	perform lookup

B.RemoteFileSlave	SendPacket(dest=A, result = done)
B+A.MagnetDrivers	B.transmit and A.receive packet
A.RemoteFileMaster	copy packet.result => file.result
A.user	continue

The structure of the FileCommand procedure of the Magnet driver is the following:

```
PROCEDURE FileCommand(VAR f : File);
  VAR filePacket : FilePacketPtr;
BEGIN
  WITH f DO
    check the command and the file for consistency;
    IF command IN {create,open} THEN
      assign a new local descriptor;
      open a port inside of local descriptor;
      open connection to the remote station
    END;
    WITH localDescriptor DO
      GetEmptyPacket(filePacket);
      WITH packet DO
         copy info from 'f' and local descriptor
              to 'filePacket';
         if necessary copy file data to the packet;
         TransmitAndReceive (desc.connection,
                             filePacket, filePacket);
         IF received packet is OK THEN
           copy info from 'packet' to 'f';
           if necessary copy data from packet to 'f'
         END
      END;
      ReturnEmptyPacket(filePacket)
    END;
  END
END FileCommand;
```

The process FileSlave has the following structure:

```
PROCESS FileSlave;
  VAR filePacket : FilePacketPtr; connId : ConnectionId;
BEGIN
  LOOP (* forever *)
    Receive(fileSlavePort, connId, filePacket);
    WITH packet DO
      IF packetType is OK THEN
        IF command IN {create, open} THEN
          assign a new local descriptor
        END;
        WITH localDescriptor DO
          copy info from packet to localDescriptor.file;
```

```
            if necessary copy data from packet
                 to localDescriptor.file;
            activate either FileCommand or DirectoryCommand;
            copy info from file and local descriptor
                 to packet;
            if necessary copy file data to the packet;
            exchange station and port addresses;
            Transmit(connId, filePacket)
          END
        ELSE ReturnEmptyPacket(filePacket)
        END
      END
    END
  END FileSlave;
```

4.5.2.1. File Transfer Protocol

The protocol used for packet communication is very simple, since the passive Ethernet can neither duplicate packets nor change the sequence of arriving packets. Although the probability that packets will arrive correctly is very high, there is a small possibility that a packet may be lost.

The protocol between the *FileMaster* and *FileSlave* is based on answering every request packet from the master by a reply from the slave. The master will not send any further packets (exept in case of a timeout) until a reply has arrived from the slave.

Sequence numbers and a timeout mechanism are provided to recognize lost packets. Each packet contains a sequence number computed modulo 32768. The request packets are sent from the master to the slave, and the slave responds with a packet with the same sequence number, the same packet type, and the same local identifications of the file. The master accepts only answers, where the above information is correct. Wrong packets are discarded and the request packet is retransmitted. A timeout mechanism is provided to supervise lost packets. Packets received more than once are filtered. The above function is provided by the procedure *TransmitAndReceive* from the module *Connection*.

The slave keeps a local copy of the result of the last operation. When a packet is received with a sequence number less than the expected one, the last reply was lost. The slave copies the last result of the operation to a packet and transmits it back. When the sequence number in the packet corresponds to the expected number, the local device driver is activated and a packet with the result of the operation is returned to the master.

The slave processes one request after the other in the sequence as they are delivered from the slave port. There is no overlapping of network and disk activities. This serialized way allows a number of users to have shared access to one slave with virtually no handling of concurrency at the slave level. The current version of remote files does not support a protocol controlling data consistency over several file accesses. For such application a higher (e.g. locking) protocol is needed.

4.6. Remote Procedure Call

The third method used to access the network is called *Remote Procedure Call*. It is based on a principle similar to the idea of remote files. The access to a *file* is done either locally (local access), or parameters of the operation are transmitted to a remote station, the operation is executed there, and the result of the operation is transmitted back (remote access).

A procedure call may be either executed locally - this is the normal way as known in all high-level programming languages. The other possibility to execute a procedure call is to send a packet containing the identification of a procedure and its parameters to a remote machine, let this machine execute the procedure, and send a packet with a result back to the calling machine.

The implementation of this concept is heavily influenced by the separate compilation facility of the language Modula-2. Remote procedures are declared in the same way as local procedures in a *definition module*. Only procedures with some restricted properties are allowed. The parameters must not be of pointer type and no global variables residing in a non-local machine must be accessed. The pointer types are excluded because of difficulties connected with dereferencing necessary to access the actual data. Global variables known to different stations would have to be copied every time they are accessed. The cost of such copying is prohibitive. For this reason no common global variables are allowed.

For one definition module with remote procedures, three different implementation modules exist: one on the calling station and two at the called station. The procedures of the module in the calling station accept the procedure calls, transform their parameters into packets, send these packets to the remote station, wait for a response, and return back to the user.

At the remote station these packets are received by a *reciprocal* module that calls the *actual* procedure with the parameters copied from a packet. After the return from the procedure, the results of the call are copied into a packet and sent back.

Only the module with the *actual* procedures must be programmed by hand. These procedures perform the actual operation. The *calling* and *reciprocal* module are automatically generated by a preprocessor reading the symbol files of a definition module and producing a Modula-2 source file.

This project is in an early stage of development. For this reason, no details about the implementation are given in this paper.

5. Network Services

On top of the basic Magnet system a number of service programs have been implemented. Since most of these services will be described in papers appearing later, only a short description of each service is given here.

5.1. Program magnetFiles

The program *magnetFiles* enables access to remote files from programs working with files but not explicitly importing any Magnet modules. The program *magnetFiles* collects all necessary Magnet modules (both the *master* and the *slave*)

and starts the command interpreter on top of it. In this way, the current virtual machine of the Medos-2 operating system is enhanced by a new device: the remote file.

The name of a user disk is specified in a designated entry in the file *User.Profile*. This file is used in the Medos-2 environment to keep user dependent parameters of service programs (e.g. the default name of a font for the editor). When the program *magnetFiles* is started, the name of a disk is installed in the network. A user of a remote station may now connect this name with a freely chosen local medium name - called *nickname (see 4.5.1.3)*. When the connection was successfully established, the remote files are accessed in the same way as local files by insering the nickname as medium name in front of a the file name.

The programs running most often on top of *magnetFiles* are *copy* and *debug*. The application of a debugger has some interesting aspects and should be considered here in more detail.

The program *debug* is used to analyze a memory dump that is produced when a user program was aborted due to a run-time error. The debugger reads this dump file and produces human readable information by applying information gained from *reference* and *listing files* produced during compilation.

When several persons work on a common project such *reference files* are located on different disks. A user may debug only private parts of programs where the debugging information is available. To enable debugging of the entire program, all reference and listing files must be copied to a single disk. Such a solution is impractical. The disk keeps a large number of unnecessary files that might even be outdated and do not correspond to the most recent version of the program. Using *magnetFiles*, a better solution is achieved. The debugger is started on top of *magnetFiles* and the reference files are fetched through the remote file access. There are no private copies of files, and the version always corresponds to the actual version of the program.

5.2. Remote Printer Prisma

The Magnet network provides access to a laser beam printer Cannon LBP10. One Lilith computer is reserved as a server of this printer. Users of Liliths access the server from their personal computer through Magnet.

The communication between a user station and a printer is based on the remote file protocol. The file to be printed is first formated on the user station. Such a formatting is necessary to transform the different file formats used by different editors to a common printer file format. Next, the formatted file is copied via Magnet by the remote file protocol to the device *PR* residing in the printer server station. This device acts as a spooling system, that copies the printer file onto a local disk, and generates a queue entry for the printer process. This process fetches entries from the input queue and prints corresponding files as they are read from a local disk.

5.3. MailBox

The mailbox system provides services enabling an exchange of electronic mail. A user sends letters either to another user or to a group of users. Another user may get a directory of his mailbox, choose one letter from it, and copy this letter either

onto his screen or onto a file.

Access to the mailbox is based on the following principle. Every operation is considered to be a finalized transaction. In a session where the user performs several operations (getting a directory, reading one letter, sending another letter, etc), the mailbox system does not keep track of previous operations. This solution was chosen to minimize the problems with inactive sessions, that do no exchange any messages for a longer time. In such a situation the server must decide either to consider the session as aborted and clean it up, or to wait for a possible restart of the activity.

The dialog between the user and server is based on the master-slave relation where the server is always the master. In this way, the server has the possibility to supervise timeouts and abort sessions in case where no responses from the user arrive.

The communication is performed in three phases. In the *request phase* the user sends an *attention packet* to the server. The server takes over the control and asks the user which action should be performed. The user answers with a packet specifying the action. The server now performs the action. If any data exchange between server and user is needed (e.g. copying of a letter), the remote file protocol is used in the *data exchange phase*. In the *end phase* the server sends a reply packet to the user containing information about the result of the action. This packet must be confirmed by the user.

5.4. Central File Store

The central file server provides access to a large disk storage. The Fujitsu disk drive M2351A with a capacity of 470 MByte and an average access time of 30 msec is used. This central file server allows concurrent access by many users, each having his private directory. For the data transfer the remote file protocol is used.

5.5. Future Services Considered for Implementation

The following services and programs are considered for future implementation:

time/date : this service returns information with the current time and date

bootstrap : this service enables booting of a Lilith computer from Magnet instead from the local disk

clearing house : this service has a small database with information about the users and services of the system. It provides operations to identify names of user and services, and to authenticate their passwords.

distributed loader : this loader enables loading and executing of programs which are stored on remote disks. This extension needs only a small change in the searching strategy of the current loader, since the loader uses the remote file system.

remote execution : when a user leaves his computer he may set it into a *listen* mode in which the station is available for use by other stations. Another user may acquire this station and download his program into this station for execution. In this way, a distributed program using a number of processors may be executed.

picture animation : the remote execution may be used for programs running a distributed picture animation. A number of processors compute the next picture and send it on demand to the master computer. Another packet format will be used for the picture transfer, since the hardware allows to transmit the complete picture (about half a million bits) in just one packet.

access to other networks : In the future there will be more Magnet networks on the campus. These Magnets will be interconnected by a slow link (9.6kbit/sec) using the local campus network *Kometh*.

6. Measurements

The basic Magnet software up to the module *Connections* needs 5428 words (decimal) of memory. This figure does not include the working space of processes and the memory used to keep the envelopes. The above entities are allocated from the heap. The storage requirement of the individual modules is as follows:

```
Nucleus      998
MagnetBase  1638
MagnetIO     360
Connections 2416
```

The module *RemoteFileMaster* requires 1657 words plus 139 words for each file that can be opened. The module *RemoteFileSlave* needs 1072 words.

The speed of the network was measured when exchanging packets at the level of *Connections*. Two processes *Master* and *Slave* were used. The *Master* produced packets and sent them by *TransmitAndReceive* to the slave, the *Slave* received them by *Receive* and sent them back by *Transmit*. The packet had a data length of 128 words with a header of 9 words. The achieved data rate was 65 kbytes/sec.
The speed of the remote file protocol was measured by reading of a sequential file. The data transfer rate is 10 kbyte/sec. For comparison, the same file was read by a local disk access. The local access data rate is about 40 kbyte/sec. It corresponds to a transfer of three sectors per disk revolution. This speed is given by the interleaving factor of 12 sectors in a track having 48 sectors. The speed of the remote access corresponds to one sector per 1.3 revolutions. The remote file access is too slow to catch the next sector, so that one full disk revolution is lost.

The long time average utilisation of the cable is neglectable, the cable is most of the time idle. During the file transfer between two stations the utilization is between 1% to 2%, some simple hardware tests utilize about 3% of the cable capacity. This measurement confirms the assumption that the software and not the hardware is the bottleneck of the system.

7. Conclusion

The Magnet network was installed at the Institute for Informatics in the fall of 1982. Over 10 stations were connected in the first phase, in summer 1983 another 20 stations were connected. At the beginning, only the program *magnetFiles* was available allowing a simple exchange of files. The utilization of the cable was at that time virtually zero. The availability of the network triggered a development of a

number of service programs. Since a printer, an electronic mail system, and a file server keeping the most recent version of system programs were installed, Magnet started to be a really useful tool for all members of the institute.

The basic system, consisting of both hardware and software parts (without application programs), was developped in average by one and half men during 30 months. Since in the university environment the main assigment is teaching and only a little time is left for research, we may assume that the whole project needed about one and half man-year.

In the following part we like to summarize our experiences gained during the development of Magnet:

The interfacing of Lilith was a relatively straight forward task, since the bus structure of the computer is rather simple. There were no serious problems either with the DMA, or with the bus access, or with interrupts. The only essential problem we encountered was the limited storage size of Lilith. The size of all modules needed for implementation of the printer server exceeded the size of the available memory. We spent some time cutting off parts of programs to fit the whole system into the machine at costs of limited security and flexibility.

The language Modula-2 was an ideal tool for programming. Especially the concepts of separate compilation influenced significantly the design of Magnet. The possibility of programming at low levels of the system allowed that the entire system, inclusive Nucleus and drivers, were programmed in a high-level language. The whole project could not have been developped in such a short time if as programming tool an assembly language or Fortran were used.

We would report only positively about the operating system Medos-2 if concurrency would be supported. The introduction of a home-grown Nucleus did not solve all problems, since the lower levels of Medos-2 make sometimes other assumptions about the behaviour of the user program then Nucleus does. Especially in case of an error, both systems behave differently, causing some situations where only the booting of the machine resolves the problem. We hope that the new version of Medos-2 will solve all these problems.

An important feature of the hardware design was the possibility that a single interface was able to transmit packets and receive them immediately. This feature allowed nearly all testing of the software to be done on a single computer. Only in the final phase more machines were used to confirm the previous testing. This feature accelerated significantly the testing and debugging, since always all debugging information was available on a single machine and very seldom several stations were needed for a test session.

The development of the software was not always a straight forward process since unfortunately some important concepts were realized too late. In various phases of the development we had to reorganize the structure of the whole system. The following example shows one such change: At the early stage of the project we liked the idea of having the entire declaration of the type *Packet* in the low level module *MagnetBase*. Such a declaration contained in a large *case-variant-record* all fields necessary for all communications on Magnet. The fields for the name server, remote file access, connections, etc. were already included in the declaration. Such a declaration allowed a very nice structuring of the system and an extensive possibility of compiler control. Problems occured when a small change

was done in the declaration of a packet. The definition module had to be changed and recompiled. After such a change all higher modules had to be recompiled even when they were not directly influenced by the change. We had to leave the policy of keeping the entire declaration in a low-level module and introduced the more flexible but less secure concept described in 4.1.3.

The problems of recovery from errors are not discussed in details in this report. Every level of software provides some functions providing facilities for error recovery of higher levels. An example could be retrieving of lost envelopes (4.1.4), resolving a deadlock situation in cases of mutually blocked envelopes, or checking of activity of a connection (4.2). These problems will be discussed in a later appearing paper.

We are very often asked, why the *old* 3 Mbit/sec version of the Ethernet was chosen and not the new 10 Mbit/sec version. When the project was started in 1980, there were no LSI chips for the 10 Mbit Ethernet available. The controller had to be built using discrete components resulting in a interface with a very large size and much higher price than for a 3 Mbit version. Unfortunately the higher speed of the new Ethernet cannot be any more handled by the conventional TTL logic, some parts must use a faster ECL technology, which requires another supply voltages. Another problem of the 10 Mbit Ethernet is the higher memory load. During a transmission of a packet a memory accessis needed every 1.6 microseconds. If a full duplex operation (the transmitted packet is immediately received back) is allowed, the memory must be accessed every 0.6 microsecond. This is faster than the Lilith memory allows, so that an on board buffering of data would be included.

8. Acknowledgements

I would like to thank here all colleagues who helped me install and implement the Magnet network. My special thanks are due to Andy Bertholsheim from Stanford University for his help with the hardware design. Beat Frey helped me very much with the hardware debugging and he designed toghether with Herbert Ruckstuhl the mailbox system. The name server and remote messages were implemented by Bernhard Wagner, Leo Geissmann implemented the printer server. Svend Erik Knudsen's excellent design of the operating system *Medos* enabled the introduction of remote files without any significant problems. Many thanks are due to Prof. N. Wirth, who initiated the whole project.

9. Literature

[Etr82] The Ethernet, A Local Area Network,
Digital Equipment Co, Intel Co, Xerox Co, 1982

[Hpr78] A. Hopper:
Local Area Communications Networks
Ph.D. Thesis, University of Cambridge 1978

[Knu83] S. E. Kudsen:
Medos-2, a Modula-2 Oriented Operating System
ETH Dissertation, ETH Zürich, 1983

[Met76] R.M. Metcalfe, D.R. Boggs :
Ethernet: Distributed Packet Switching for Local Computer Networks,
Communication ACM July 1976

[Som76] R. Sommer:
COBUS, a Firmware Controlled Data Transmission System
EUROMICRO, North-Holland 1976

[Wir81] N. Wirth :
The Personal Computer Lilith,
Software development Environments, A.I. Wasserman, Ed.
IEEE Computer Society Press, 1981

[Wir82] N. Wirth :
Programming in Modula-2, Springer 1982

Appendix

Definition Modules

The description of the Magnet objects used in this report does not always exactly correspond to their declaration in definition modules. For reason of clarity some objects are not mentioned at all, for some other objects different names are used that could be better understood by the reader, or some procedures have a simplified parameter list. The following appendix lists the definition modules of Nucleus, MagnetBase, MagnetIO, Connections, RemoteFileMaster, and RemoteFileSlave. The definition module for the remote messages is not included, since it is still under development and some changes may be expected. It will be published in a later appearing report.

```
(*******************************************)
(*                                         *)
(*                                         *)
      DEFINITION MODULE Nucleus;
(*                                         *)
(*          Author Jirka Hoppe             *)
(*       Institut fuer Informatik          *)
(*             ETH Zurich                  *)
(*            Switzerland                  *)
(*                                         *)
(*           version 8/12/82               *)
(*                                         *)
(*******************************************)

FROM SYSTEM IMPORT ADDRESS;
EXPORT QUALIFIED
(*TYPE*)         Port,  MsgPointer, PortPointer,
(*VAR*)          nilPort, debugPrint,
(*PROCEDURE*)    CreateProcess, SendMsg, WaitMsg, WaitMsgTimeOut,
                 MessageCall, WaitIO, Pause, CreateMsg, GetInfoPointer,
                 GetInfoSize, GetAnswerPort, InitPort, QueuedMsg,
                 StartSystem, PointsToNilPort, GetOwnerProcess, ProcessId,
                 IsMainProcess;


TYPE   ProcPointer;  (* hidden *)
       MsgPointer;   (* hidden *)
       PortPointer = POINTER TO Port;
       devIndex    = [7..15];

TYPE  Port   = RECORD
                 (* pointer to the next waiting process *)
                 nextWProc : ProcPointer;
                 (* pointer to the next waiting message *)
                 nextWMsg  : MsgPointer;
               END;


VAR nilPort        : Port;    (* will be used as a dummy return address  *)
    debugPrint     : BOOLEAN; (* TRUE =>  debug info will be displayed *)

PROCEDURE CreateProcess( ProcessCode : PROC; (* this procedure will be a process *)
(*----------------------*) stackSize   : CARDINAL; (* size of the stack *)
                           VAR done    : BOOLEAN);
(* creates a process from the procedure 'ProcessCode'
   and assigns a stack with 'stacksize' to it,
   'done' specifies if the creation was succesfull *)
```



```
PROCEDURE CreateMsg( VAR msg  : MsgPointer; (* message to be created *)
(*-----------------*) sizeInfo : CARDINAL ;  (* size of the information *)
                     VAR done : BOOLEAN);
(* assignes the necessary amount of storage from the current heap,
   initializes the hidden information and returns a pointer to the message,
   done = TRUE if everything was OK *)

PROCEDURE StartSystem(keepMain : BOOLEAN);
(*--------------------*)
(* after all initialization has been done: start system
   If keepMain = TRUE then the initializing prozess is introduced
   as a process as well *)

PROCEDURE InitPort( VAR port : Port);
(* ---------------- initializes the port *)

PROCEDURE SendMsg( VAR port       : Port;  (* send msg to this port *)
(*---------------*) VAR answerPort : Port;  (* sender of the message *)
                   VAR msg        : MsgPointer );
(* sends the message 'msg' to the 'port'
   'answerPort' specifies where the answer should be sent,
   if no 'answerPort' is needed => use the 'nilPort',
   if there is a process waiting in the 'port' it will be resumed,
   if there is no process, the message will be queued *)

PROCEDURE WaitMsg( VAR port : Port;        (* get message from this port *)
(*---------------*) VAR msg  : MsgPointer );
(* receives a message from 'port',
   if there is no message in the 'port', the calling process is delayed.
   the procedure 'GetAnswerPort' may be used to find the port
   where the answer message should be sent *)

PROCEDURE WaitMsgTimeOut( VAR port    : Port; (* get message from this port *)
(*---------------------*)  VAR msg     : MsgPointer ;
                            timeOut : CARDINAL;
                          VAR ok      : BOOLEAN);
(* similar to WaitMsg, but a timeout in multiples of 20msec
   may be specified,
   ok=TRUE if there was no time out *)

PROCEDURE MessageCall( VAR port   : Port;       (* receiver port *)
(*------------------*)  VAR request: MsgPointer; (* send from me to the
                                                    other process *)
                       VAR answer : MsgPointer); (* answer from
                                                    the other process *)
(* corresponds to the pair
     SendMsg(port, localPort, request);
     WaitMsg(localPort, answer); *)

PROCEDURE WaitIO(devNr : devIndex );
(*-------------*)
(* delays a process until interrupt from the device 'devNr' occurs *)

PROCEDURE Pause(ticks : CARDINAL);
(*-------------*)
(* delays a process until ticks*20msec
   if 'ticks' is 0 => the calling process stays ready
   but another ready processes will be started *)

PROCEDURE QueuedMsg( port : Port) : CARDINAL;
(*-----------------*)
(* returns the number of waiting messages in the port *)

PROCEDURE GetAnswerPort(msg : MsgPointer): PortPointer;
(*--------------------*)
(* returns the port where the answer should be sent
   this port is specified in the procedure 'SendMsg'
   procedure 'PointsToNilPort' may be used to find if the 'nilPort' was used *)
```

```
PROCEDURE PointsToNilPort(pP : PortPointer) : BOOLEAN;
(*-----------------------*)
(* TRUE if the pP points to the nilPort *)

PROCEDURE GetInfoPointer(msg : MsgPointer): ADDRESS;
(*-----------------------*)
(* returns the pointer to the user part of message *)

PROCEDURE GetInfoSize(msg : MsgPointer): CARDINAL;
(*-------------------*)
(* return the size of the user part of message *)


(*---- the following part is used for cleaning up and debugging only -----*)

PROCEDURE GetOwnerProcess(msg : MsgPointer) : ADDRESS;
(*-----------------------*)
(* returns the current owner of a message  *)

PROCEDURE ProcessId():ADDRESS;
(*-----------------*)
(* gives the identification of the current process *)

PROCEDURE IsMainProcess(id : ADDRESS) : BOOLEAN;
(*---------------------*)
(* returns TRUE if the id (returned from ProcessId) is the main process *)

END Nucleus.
```



```
(*****************************************)
(*                                       *)
(*              M A G N E T              *)
(*                                       *)
      DEFINITION MODULE MagnetBase;
(*                                       *)
(*          Author Jirka Hoppe           *)
(*      Institut fuer Informatik         *)
(*            ETH Zurich                 *)
(*           Switzerland                 *)
(*                                       *)
(*          version 18/4/83              *)
(*                                       *)
(*****************************************)

FROM SYSTEM     IMPORT ADDRESS;
IMPORT Nucleus;

EXPORT QUALIFIED
(*CONST*)       maxActivePort,
(*TYPE*)        PortNumbers, Reply, Packet, PacketPointer, Envelope, Port,
(*VAR*)         entryPort, xmitReq, mDebug,
(*PROCEDURE*)   LocalStatAdr, OpenMagnetPort, CloseMagnetPort, PortOpened,
                OpenFilterAddress, CloseFilterAddress,
                GetEmptyEnvelope, ReturnEmptyEnvelope, LetOtherWork,
                MagnetBaseLevel, StopMagnet, MaxUserTypeSize,
                GetEnvelopePointers;
(*-----------*)
CONST
  maxActivePort    =  15; (* MAGNET can handle so many predefined opened ports *)


TYPE
Envelope    = Nucleus. MsgPointer; (* packets are sent inside of a Envelope *)
Port        = Nucleus.Port;        (* used for addressing *)
PortNumbers = [0..maxActivePort];  (* predefined ports *)


Reply = (pOk, pTimeOut, tooManyRetries, softXsumEr, hardXsumEr, lengthError,
         aborted, dmaOver, overRun, badConnection, otherErr);

Packet =
  RECORD
    destAdr    : CARDINAL; (* must be the first word in the packet,
                              destination address *)
    xsum       : BITSET;   (* soft checksum of the packet; 2nd field in the packet*)
    srcAdr     : CARDINAL; (* source address *)
    destPort   : ADDRESS;  (* destination port for the packet *)
    returnPort: ADDRESS;   (* reply port *)
    connKeys   : CARDINAL; (* packed identifiers of the connection *)
    seqNr      : CARDINAL; (* sequence number *)
    pLength    : CARDINAL; (* length of the packet *)
    reply      : Reply;
    (* the next field is overlaid by be the user specified type
       it can be accessed by the 'userInfo' pointer
       specified in various procedures *)
    packetType : CARDINAL;
  END;

  PacketPointer = POINTER TO Packet;

(*----------------*)

VAR
  entryPort     : ARRAY PortNumbers OF Port; (* distributor of received packets
                                                predefined ports only*)
  xmitReq       : Port;                    (* port for transmitter requests *)
  mDebug        : BOOLEAN;                 (* debugging switch *)
```

```
(*----------------*)

PROCEDURE OpenMagnetPort(portId: ADDRESS; VAR done : BOOLEAN);
(* ---------------------- open a port to receive packets
   portId is either a small integer indicating predefined ports
   or a pointer to a Port *)

PROCEDURE CloseMagnetPort(portId: ADDRESS; VAR done : BOOLEAN);
(* ---------------------- close a port
   portId is either a small integer indicating predefined ports
   or a pointer to a MagnetPort*)

PROCEDURE PortOpened(portId : ADDRESS) : BOOLEAN;
(*---------------- TRUE if this port is opened for magnet services *)

PROCEDURE GetEmptyEnvelope(VAR envelope : Envelope;
(*-----------------------*) VAR packet   : PacketPointer;
                            VAR userInfo : ADDRESS;
                                maxSize  : CARDINAL);
(* gets an empty envelope from the 'emptyPackets' port,
   assignes the 'packet' to a pointer to the packet subfield
   and 'userInfo' to a pointer to the user part of packet
   maxSize specifies the size of the user part of the packet=> use TSIZE(userType)
*)

PROCEDURE ReturnEmptyEnvelope(VAR envelope : Envelope);
(*------------------------ returns an envelope back to the emptyPacket port*)

PROCEDURE GetEnvelopePointers( envelope     : Envelope;
(*--------------------------*) VAR packet   : PacketPointer;
                               VAR userInfo : ADDRESS);
(* returns the pointers to the content of the envelope *)

PROCEDURE OpenFilterAddress(n : CARDINAL);
(*-----------------------*)
(* opens the hardware address filter for packets with a new address *)

PROCEDURE CloseFilterAddress(n : CARDINAL);
(*-------------------------*)
(* closes a hardware address filter of a station *)

PROCEDURE LetOtherWork;
(*------------------- releases the processor in favour of other processes *)

PROCEDURE MaxUserTypeSize(): CARDINAL;
(*-----------------------*)
(* returns the size of the user specified part of packet *)

PROCEDURE LocalStatAdr() : CARDINAL;
(* ---------------- returns the address of the station as read from the hardware *)

PROCEDURE MagnetBaseLevel(): CARDINAL;
(*------------------------- returns the declaration level of MagnetBase *)

PROCEDURE StopMagnet;
(*------------------ turns off the magnet interface *)

END MagnetBase.
```



```
(*****************************************)
(*                                       *)
(*              M A G N E T              *)
(*                                       *)
      DEFINITION MODULE MagnetIO;
(*                                       *)
(*         Author Jirka Hoppe            *)
(*      Institut fuer Informatik         *)
(*            ETH Zurich                 *)
(*           Switzerland                 *)
(*                                       *)
(*          version 18/4/83              *)
(*                                       *)
(*****************************************)

(* some routines making the use of the cable more comfortable *)
FROM SYSTEM     IMPORT WORD, ADDRESS;
FROM MagnetBase IMPORT Port, Reply, PacketPointer, Envelope;

EXPORT QUALIFIED
(*CONST*)  seqNrModulo, retryFlag, CommDesc, noXsum,
(*TYPE *)  DisplayProc,
(*PROC *)  Transmit, Receive, TransmitAndReceive,
           InstallDisplayProc, RemoveDisplayProc, Read;

CONST
  noXsum      = {0..15}; (* this xsum is always ok for 'Receive' *)
  seqNrModulo = 100000b; (* sequence numbers are computed MOD seqNrModulo *)
  retryFlag   = 0;       (* This bit is set into the sequence number
                            if the packet is transmitted twice *)
TYPE
  CommDesc = RECORD     (* control of the communication by TransmitAndReceive *)
               portId  : ADDRESS;  (* ADR(port)*)
               sqNr    : CARDINAL; (* sequence Nr of the packet *)
                                   (* sqNr < packet.seqNr => ignore *)
                                   (* sqNr > packet.seqNr => ignore but retransmit*)
               retries : CARDINAL; (* nr of retries *)
               timeOut : CARDINAL; (* timeout interval between retries *)
             END;

   DisplayProc = PROCEDURE(CARDINAL, ADDRESS);
   (* display the content of user part of a packet; used for debugging only
      CARDINAL = packetType, ADDRESS = pointer to the user part *)


PROCEDURE Transmit(VAR envelope : Envelope; (* envelope of packet
(*----------------*)                          to be transmitted *)
                   VAR reply    : Reply);
(* transmit a packet, compute soft xsum, all other fields must be set;
   packet.pLength is the size of the user part of packet
   after the transmission the packet is returned to the empty packet pool,
   reply designates the success  *)

PROCEDURE Receive(VAR port     : Port;     (* wait for packets for this port *)
(*---------------*)VAR envelope : Envelope; (* envelope to be received *)
                   VAR packet   : PacketPointer; (* points to the enclosed received
                                                    packet *)
                   VAR userInfo : ADDRESS;  (* points to the user part of packet *)
                       timeOut  : CARDINAL;
                       maxSize  : CARDINAL );
(* waits until the packet to 'port' arrives OR the timeout expires
   packet is assigned to the packet part of the envelope
   the 'userInfo' points to the user part of the packet
   xsum is recomputed and checked;
   timeout = 0 => waits forever
   maxSize is the maximal size of the user part => use TSIZE(userType) *)
```

```
PROCEDURE TransmitAndReceive
(*------------------------*)
    (    commDesc: CommDesc;      (* descriptor of communication*)
     VAR xmitEnv : Envelope;      (* envelope to be transmitted *)
     VAR recEnv  : Envelope;      (* recieved envelope *)
     VAR packet  : PacketPointer; (* points to to enclosed received packet *)
     VAR userInfo: ADDRESS;       (* points to the user part of packet *)
         maxSize : CARDINAL );

(* corresponds to
   REPEAT
     Transmit(xmitEnv,reply)
     Recieve(portId, recEnv, timeOut)
   UNTIL (reply=pOk) AND (recPacket.packetType=pType) OR (too many retires) *)

PROCEDURE InstallDisplayProc(type : CARDINAL; proc : DisplayProc;
(* ---------------------- *) VAR done : BOOLEAN);
(* install display procedure for debugging for the packet type 'type' *)

PROCEDURE RemoveDisplayProc(type : CARDINAL);
(*-------------------------*)
(* remove display procedure for the packet type 'type') *)

PROCEDURE Read( VAR ch : CHAR);
(*------------ the same as Terminal.Read but with a better waiting *)

END MagnetIO.
```



```
(*****************************************)
(*                                       *)
(*              M A G N E T              *)
(*                                       *)
      DEFINITION MODULE Connections;
(*                                       *)
(*  Authors Jirka Hoppe and Bernhard Wagner  *)
(*      Institut fuer Informatik         *)
(*            ETH Zurich                 *)
(*           Switzerland  *              *)
(*                                       *)
(*          version 23/2/83              *)
(*                                       *)
(*****************************************)

FROM SYSTEM  IMPORT ADDRESS;
FROM Nucleus IMPORT Port;
IMPORT MagnetBase1;

EXPORT QUALIFIED
(*CONST*)       seqNrModulo,
(*TYPE*)        ConnectionId, PacketParam, Result, Reply,
(*PROCEDURE*)   OpenConnection, CloseConnection, ConnectionActive,
                InstallService, RemoveService,
                GetEmptyPacket, ReturnEmptyPacket,
                Transmit, Receive, TransmitAndReceive;

CONST
  seqNrModulo    = 100000b; (* limit for sequence number *)

TYPE
  ConnectionId; (* hidden - an identification of a connection *)

  Reply  = MagnetBase.Reply;

  PacketParam = RECORD
                  length : CARDINAL; (* length of packet          IN/OUT *)
                  seqNum : CARDINAL; (* sequence nummer           IN/OUT *)
                  repl   : Reply;    (* reply from packet         OUT *)
                  timOut : CARDINAL; (* time to wait for packets  IN *)
                  retry  : CARDINAL; (* nr of retries             IN *)
                END;

  Result = (connOk, connNotOk, connParamErr, connFull, connOtherErr);


(* ------   procedures handling connections ------  *)

PROCEDURE OpenConnection(     partnerName : ARRAY OF CHAR;
(*----------------------*)      localPortId : ADDRESS;
                          VAR connId      : ConnectionId;
                          VAR serviceInfo : ARRAY OF CHAR;
                          VAR result      : Result);

PROCEDURE CloseConnection(    connId      : ConnectionId;
(*----------------------*)  VAR result      : Result);

PROCEDURE ConnectionActive(connId         : ConnectionId):BOOLEAN;
(*-----------------------*)

PROCEDURE InstallService(   localName     : ARRAY OF CHAR;
(*----------------------*)    localPortId   : ADDRESS;
                              serviceInfo   : ARRAY OF CHAR;
                          VAR result        : Result);

PROCEDURE RemoveService(     localName    : ARRAY OF CHAR;
(*----------------------*) VAR result      : Result);
```

```
(* ----- flow of empty packets ----- *)

PROCEDURE GetEmptyPacket(VAR userPtr : ADDRESS; maxSize : CARDINAL);
(*-----------------------*)

PROCEDURE ReturnEmptyPacket(VAR userPtr : ADDRESS);
(*-----------------------*)

(* ----- exchange of packets via Magnet ----- *)

PROCEDURE Transmit(    connId     : ConnectionId;
(*----------------*)    userPtr    : ADDRESS;
                   VAR packetPar  : PacketParam);

PROCEDURE Receive( VAR port       : Port;
(*---------------*) VAR connId     : ConnectionId;
                   VAR userPtr    : ADDRESS;
                   VAR packetPar  : PacketParam;
                       maxSize    : CARDINAL);

PROCEDURE TransmitAndReceive(    connId      : ConnectionId;
(*-------------------------*)    xmitUserPtr : ADDRESS;
                             VAR recUserPtr  : ADDRESS;
                             VAR packetPar   : PacketParam;
                                 maxSize     : CARDINAL);

END Connections.
```



```
(*******************************************)
(*                                         *)
(*              M A G N E T                *)
(*                                         *)
      DEFINITION MODULE RemoteFileMaster;
(*                                         *)
(*          Author Jirka Hoppe             *)
(*      Institut fuer Informatik           *)
(*             ETH Zurich                  *)
(*            Switzerland                  *)
(*                                         *)
(*           version 11/4/83               *)
(*                                         *)
(*******************************************)

EXPORT QUALIFIED  InstallRemoteMedium, RemoveRemoteMedium;


PROCEDURE InstallRemoteMedium
(*-------------------------*)
  (localMedium   : ARRAY OF CHAR; (*name of local medium*)
   partnerMedium : ARRAY OF CHAR; (*name of remote medium*)
   timeOut       : CARDINAL;      (*maximal time to wait for one response
                                    from remote medium *)
   VAR done      : BOOLEAN);
(* install the Magnet Remote System Master as a device in the local file system
   the device is called locally 'localMedium' (eq. XY)
   the remote device is called 'partnerMedium' (eq. JohnOK )  *)

PROCEDURE RemoveRemoteMedium
(*-------------------------*)
  (localMedium  : ARRAY OF CHAR; (*name of the local medium*)
   VAR done     : BOOLEAN);

END RemoteFileMaster.




(*******************************************)
(*                                         *)
(*              M A G N E T                *)
(*                                         *)
   DEFINITION MODULE RemoteFileSlave;
(*                                         *)
(*          Author Jirka Hoppe             *)
(*      Institut fuer Informatik           *)
(*             ETH Zurich                  *)
(*            Switzerland                  *)
(*                                         *)
(*           version 12/11/82              *)
(*                                         *)
(*******************************************)
EXPORT QUALIFIED  readOnly;
  VAR readOnly : BOOLEAN; (* =TRUE => no write access is allowed *)
END RemoteFileSlave.
```

Berichte des Instituts für Informatik

1978

*Nr.25	U. Ammann:	Error Recovery in Recursive Descent Parsers and Run-time Storage Organization
Nr.26	E. Zachos:	Kombinatorische Logik und S-Terme
*Nr.27	N. Wirth:	MODULA-2

1979

*Nr.28	J. Nievergelt, J. Weydert:	Sites, Modes and Trails: Telling the User of an Interactive System where he is, what he can do, and how to get to places
*Nr.29	A.C. Shaw:	On the Specification of Graphic Command Languages and their Processors
*Nr.30	B. Thurnherr, C.A. Zehnder:	Global Data Base Aspects, Consequences for the Relational Model and a Conceptual Schema Language
*Nr.31	A.C. Shaw:	Software Specification Languages based on regular Expressions
Nr.32	E. Engeler:	Algebras and Combinators
*Nr.33	N. Wirth:	A Collection of PASCAL Programs

1980

*Nr.34	R. Marti, J. Rebsamen, B. Thurnherr:	Meta Data Base Design - Consistent Description of a Data Base Management System
*Nr.35	H.H. Nägeli, R.Schoenberger:	Preventing Storage Overflows in High-level Languages
	J. Hoppe:	A Simple Nucleus written in Modula-2
*Nr.36	N. Wirth:	MODULA-2 (second edition)
Nr.37	Hp. Bürkler, C.A. Zehnder:	EDV-Projektentwicklung - Ein Arbeitsheft für Informatik-Studenten
*Nr.38	H. Burkhart, J. Nievergelt:	Structure-oriented editors
*Nr.39	A. Meier, C.A. Zehnder:	Flächenmodell-Register: Die Strukturen wichtiger geographischer Datensammlungen der Schweiz

1981

*Nr.40	N. Wirth:	The Personal Computer Lilith
Nr.41	T.M. Fehlmann:	Theorie und Anwendung des Graphmodells der Kombinatorischen Logik
Nr.42	E. Graf:	Probabilistische Algorithmen und Computer-unterstützte Untersuchungen von probabilistischen Primalitätstests
*Nr.43	H. Burkhart:	Konzepte zur Systematisierung der Benutzerschnittstelle in interaktiven Systemen und ihre Anwendung auf den Entwurf von Editoren
*Nr.44	J. Nievergelt, F.P. Preparata:	Plane-sweep Algorithms for Intersecting Geometric Figures
*Nr.45	M. Reimer, J.W. Schmidt:	Transaction Procedures with Relational Parameters
Nr.46	J. Nievergelt, H.Hinterberger, K.C. Sevcik:	The Grid File: An adaptable, symmetric multi-key file structure

1982

Nr.47	J. Nievergelt:	Errors in dialog design and how to avoid them
Nr.48	P. Läuchli:	PG - Ein interaktives System für die Manipulation von Figuren der projektiven Geometrie
Nr.49	A. Meier:	A Graph Grammar Approach to Geographic Data Bases
*Nr.50	J. Rebsamen, M. Reimer, P. Ursprung, C.A. Zehnder:	LIDAS A Database System for the Personal Computer Lilith The Database Management

1983

*Nr.51	K.J.Lieberherr, S.E. Knudsen:	Zeus: A Hardware Description Language for VLSI
Nr.52	F.L.Ostler:	An SMD Disk Controller for the Lilith Computer
Nr.53	R.P.Brägger, M.Reimer:	Predicative Scheduling: Integration of Locking and Optimistic Methods
Nr.54	K.Hinrichs, J.Nievergelt:	The grid file: a data structure designed to support proximity queries on spatial objects
Nr.55	C.A.Zehnder (Ed.):	Database Techniques for Professional Workstations
Nr.56	J. Gutknecht:	System Programming in Modula-2: Mouse and Bitmap Display
Nr.57	J. Hoppe:	MAGNET: A Local Network for Lilith Computers

* out of stock