10. The Network

10.1. Introduction

Workstations are typically, but not always, connected in a local environment by a network. There exist two basically different views of the architecture of such nets. The more demanding view is that all connected stations constitute a single, unified workspace (also called address-space), in which the individual processors operate. It implies the demand that the "thin" connections between processors are hidden from the users. At worst they might become apparent through slower data access rates between the machines. To hide the difference between access within a computer and access between computers is regarded primarily as a challenge to implementors.

The second, more conservative view, assumes that individual workstations are, although connected, essentially autonomous units which exchange data infrequently. Therefore, access of data on partner stations is initiated by explicit transfer commands. Commands handling external access are not part of the basic system, but rather are implemented in modules that might be regarded as applications.

In the Oberon System, we adhere to this second view, and in this chapter, we describe the module *Net*, which is an autonomous command module based on the network driver SCC. It can be activated on any station connected in a network, and all of them are treated as equals. Such a set of loosely coupled stations may well operate in networks with moderate transmission rates and therefore with low-cost hardware interfaces and twisted-pair wires.

An obvious choice for the unit of transferred data is the file. The central theme of this chapter is therefore file transfer over a network. Some additional facilities offered by a dedicated server station will be the subject of Chapter 11. The commands to be presented here are a few only: *SendFiles, ReceiveFiles*, and *SendMsg*.

As explained in Chapter 2, Oberon is a single-process system where every command monopolizes the processor until termination. When a command involves communication over a network, (at least) two processors are engaged in the action at the same time. The Oberon paradigm therefore appears to exclude such cooperation; but fortunately it does not, and the solution to the problem is quite simple.

Every command is initiated by a user operating on a workstation. For the moment we call it the *master* (of the command under consideration). The addressed station - obviously called the *server* - must be in a state where it recognizes the command in order to engage in its execution. Since the command - called a *request* - arrives in encoded form over the network, an Oberon task represented by a handler procedure must be inserted into the event polling loop of the system. Such a handler must have the general form

```
IF event present THEN handle event END
```

The guard, in this case, must imply that a request was received from the network. We emphasize that the event is sensed by the server only after the command currently under execution, if any, has terminated. However, data arrive at the receiver immediately after they are sent by the master. Hence, any sizeable delay is inherently inadmissible, and the Oberon metaphor once again appears to fail. It does not fail, however, because the unavoidable, genuine concurrency of sender and receiver action is handled within the driver module which places the data into a buffer. The driver is activated by an interrupt, and its receiver buffer effectively decouples the partners and removes the stringent timing constraints. All this remains completely hidden within the driver module.

10.2. The protocol

If more than a single agent participates in the execution of a command, a convention must be established and obeyed. It defines the set of requests, their encoding, and the sequence of data exchanges that follow. Such a convention is called a *protocol*. Since in our metaphor, actions initiated by the master and the server strictly follow each other in alternation, the protocol can be defined using EBNF (extended Backus-Naur formalism), well-known from the syntax specification of languages. Items originating from the master will be written with normal font, those originating from the server appear in italics.

A simple form of the *ReceiveFile* request is defined as follows and will be refined subsequently:

ReceiveFile = SND filename (*ACK data* | *NAK*).

Here, the symbol SND represents the encoded request that the server send the file specified by the file name. ACK signals that the request is honoured and the requested data follow. The NAK symbol indicates that the requested file cannot be delivered. The transaction clearly consists of two parts, the request and the reply, one from each partner.

This simple-minded scheme fails because of the limitation of the size of each transmitted portion imposed by the network driver. We recall that module SCC restricts the data of each packet to 512 bytes. Evidently, files must be broken up and transmitted as a sequence of packets. The reason for this restriction is transmission reliability. The break-up allows the partner to confirm correct receipt of a packet by returning a short acknowledgement. Each acknowledgement also serves as request for the next packet. An exception is the last acknowledgement following the last data portion, which is characterized by its length being less than the admitted maximum. The revised protocol is defined as

ReceiveFile = SND filename (*DAT data* ACK {*DAT data* ACK} | *NAK*).

We now recall that each packet as defined in Section 9.3. is characterized by a type in its header. The symbols SND, DAT, ACK, and NAK indicate this packet type. The data portions of ACK and NAK packets are empty.

The revised protocol fails to cope with transmission errors. Correct transmission is checked by the driver through a cyclic redundancy check (CRC), and an erroneous packet is simple discarded. This implies that a receiver must impose a timing constraint. If an expected packet fails to arrive within a given time period (timeout), the request must be repeated. In our case, a request is implied by an acknowledgement. Hence, the acknowledgement must specify whether the next (normal case) or the previously requested (error case) packet must be sent. The solution is to attach a sequence number to each acknowledgement and to each data packet. These numbers are taken modulo 8, although in principle modulo 2 would suffice.

With the addition of a user identification and a password to every request, and of an alternate reply code NPR for "no permission", the protocol reaches its final form:

ReceiveFile = SND username password filename (*datastream* | *NAK* | *NPR*).
datastream = *DAT*$_0$ *data* ACK$_1$ {*DAT*$_i$ *data* ACK$_{i+1}$}.

The protocol for file transmission from the master to the server is defined similarly:

SendFile = REC username password filename (*ACK0 datastream* | *NAK* | *NPR*).
datastream = DAT$_0$ data *ACK*$_1$ {DAT$_i$ data *ACK*$_{i+1}$}.

The third request listed above, SendMsg, does not refer to any file, but merely transmits and displays a short message. It is included here for testing the link between two partners and perhaps for visibly acknowledging a rendered service by the message "done", or "thank you".

SendMsg = MSG message *ACK*.

10.3. Station addressing

Every packet must carry a destination address as well as the sender's address. Addresses are station numbers. It would certainly be inconvenient for a user to remember the station number of a desired partner. Instead, the use of symbolic names is preferred. We have become accustomed to use the partner's initials for this purpose.

The source address is inserted automatically into packet headers by the driver. It is obtained from a dip switch set when a computer is installed and connected. But where should the destination address come from? From the start we reject the solution of an address table in every workstation because of the potential inconsistencies. The concept of a centralized authority holding a name/address dictionary is equally unattractive, because of the updates required whenever a person uses a different computer. Also, we have started from the premise to keep all participants in the network equal.

The most attractive solution lies in a decentralized name service. It is based on the broadcast facility, i.e. the possibility to send a packet to all connected stations, bypassing their address filters with a special destination address (-1). The broadcast is used for emitting a name request containing the desired partner's symbolic name. A station receiving the request returns a reply to the requester, if that name matches its own symbolic name. The requester then obtains the desired partner's address from the source address field of the received reply. The corresponding simple protocol is:

```
NameRequest = NRQ partnername [NRS].
```

Here, the already mentioned timeout facility is indispensible. The following summarizes the protocol developed so far:

```
protocol =    {request}.
request =     ReceiveFile | SendFile | SendMsg | NameRequest.
```

The overhead incurred by name requests may be reduced by using a local address dictionary. In practice, a single entry is satisfactory. A name request is then needed whenever the partner changes.

10.4. The implementation

Module *Net* is an implementation of the facilities outlined above. The program starts with a number of auxiliary, local procedures. They are followed by procedure *Serve* which is to be installed as an Oberon task, and the commands *SendFiles, ReceiveFiles,* and *SendMsg*, each of which has its counterpart within procedure *Serve*. At the end are the commands for starting and stopping the server facility.

For a more detailed presentation we select procedure *ReceiveFiles*. It starts out by reading the first parameter which designates the partner station from the command line. Procedure *FindPartner* issues the name request, unless the partner's address has already been determined by a previous command. The global variable partner records a symbolic name (id) whose address is stored in the destination field of the global variable head0, which is used as header in every packet sent by procedure *SCC.SendPacket*. The variable partner may be regarded as a name cache with a single entry and with the purpose of reducing the number of issued name requests.

If the partner has been identified, the next parameter is read from the command line. It is the name of the file to be transmitted. If the parameter has the form *name0:name1*, the file stored on the server as *name0.name1* is fetched and stored locally as name1. Hence, name0 serves as a prefix of the file name on the server station.

Thereafter, the request parameters are concatenated in the local buffer variable *buf*. They are the user's name and password followed by the file name. (User name and password remain unused by the server presented here). The command package is dispatched by the call *Send(SND, k, buf)*, where *k* denotes the length of the command parameter string. Then the reply packet is awaited by calling *ReceiveHead*. If the received packet's type is DAT with sequence number 0, a new file is established. Procedure *ReadData* receives the data and stores them in the new file, obeying the protocol defined in Section 10.2. This process is repeated for each file specified in the list of file names in the command line.

Procedure *ReceiveHead(T)* receives packets and discards them until one arrives from the partner from which it is expected. The procedure represents an input filter in addition to the one provided by the hardware. It discriminates on the basis of the packets' source address, whereas the hardware filter discriminates on the basis of the destination address. If no packet arrives within the allotted time T, a type code -1 is returned, signifying a timeout.

Procedure *ReceiveData* checks the sequence numbers of incoming data packets (type 0 - 7). If an incorrect number is detected, an ACK-packet with the previous sequence number is returned (type 16 - 23), requesting a retransmission. At most two retries are undertaken. This seems to suffice considering that also the server does not accept any other requests while being engaged in the transmission of a file.

The part corresponding to *ReceiveFiles* within procedure *Serve* is guarded by the condition *head1.typ = SND*. Variable *head1* is the recipient of headers whenever a packet is received by *ReceiveHead*. First, the request's parameters are scanned. *Id* and *pw* are ignored. Then the requested file is opened. If it exists, the transmission is handled by *ReceiveData*'s counterpart, procedure *SendData*. The time limit for receiving the next request is *T1*, whereas the limit of *ReceiveData* for receiving the next data packet is *T0. T1* is roughly *T0* multiplied by the maximum number of possible (re)transmissions. Before disengaging itself from a transaction, the sender of data waits until no further retransmission requests can be expected to arrive. The value *T0* (300) corresponds to 1s; the time for transmission of a packet of maximum length is about 16ms.

Procedure *SendFiles* is designed analogously; its counterpart in the server is guarded by the condition *head1.typ = REC*. The server accepts the request only if its state is unprotected (global variable *protected*). Otherwise the request is negatively acknowledged with an NPR packet. We draw attention to the fact that procedures *SendData* and *ReceiveData* are both used by command procedures as well as by the server.

11. A Dedicated file-distribution and mail-server

11.1. Concept and structure

In a system of loosely coupled workstations it is desirable to centralize certain services. A first example is a common file store. Even if every station is equipped with a disk for permanent data storage, a common file service is beneficial, e.g. for storing the most recent versions of system files, reference documents, reports, etc. A common repository avoids inconsistencies which are inevitable when local copies are created. We call this a *file distribution service*.

A centralized service is also desirable if it requires equipment whose cost and service would not warrant its acquisition for every workstation, particularly if the service is infrequently used. A prime example of this case is a *printing service*.

The third case is a communication facility in the form of *electronic mail*. The repository of messages must inherently be centralized. We imagine it to have the form of a set of mailboxes, one for each user in the system. A mailbox needs to be accessible at all times, i.e. also when its owner's workstation has been switched off.

A last example of a centralized service is a *time server*. It allows a station's real time clock to be synchronized with a central clock.

In passing we point out that every user has full control over his station, including the right to switch it on and off at any time. In contrast, the central server is continuously operational.

In this chapter, we present a set of server modules providing all above mentioned services. They rest on the basic Oberon System without module *Net* (see Chapter 10). In contrast to *Net*, module *NetServer*, which handles all network communication, contains no command procedures (apart from those for starting and stopping it). This is because it never acts as a master. The counterparts of its server routines reside in other modules, including (an extended version of) *Net*, on the individual workstations.

Routines for the file distribution service are the same as those contained in module *Net*, with the addition of permission checks based on the received user names and passwords. Routines for printing and mail service could in principle also be included in *NetServer* in the same way. But considerations of reliability and timing made this simple solution appear as unattractive. A weaker coupling in time of data transmission and data consumption is indeed highly desirable. Therefore, data received for printing or for dispatching into mailboxes are stored (by *NetServer*) into temporary files and thereafter "handed over" to the appropriate agent, i.e. the print server or the mail server.

This data-centered interface between servers - in contrast to procedural interfaces - has the advantage that the individual servers are independent in the sense that none imports any other. Therefore, their development could proceed autonomously. Their connection is instead a module which defines a data structure and associated operators for passing temporary files from one server to another. The data structure used for this purpose is the first-in-first-out queue. We call its elements tasks, because each one carries an objective and an object, the file to be processed. The module containing the FIFOs is called *Core*. The resulting structure of the involved modules is shown in Fig. 11.1.

Fig. 11.1. includes yet another server, *LineServer*, and shows the ease with which additional servers may be inserted in this scheme. They act as further sources and/or sinks for tasks, feeding or consuming the queues contained in *Core*. *LineServer* indeed produces and consumes tasks like *NetServer*. Instead of the RS-485 bus, it handles the RS-232 line which, connetced to a modem, allows access to the server over telephone lines. We refrain from describing this module in further detail, because in many ways it is a mirror of *NetServer*.

A centralized, open server calls for certain protection measures against unauthorized use. We recall that requests always carry a user identification and a password as parameters. The server checks their validity by examining a table of users. The respective routines and the table are contained in module *Core* (see Sect. 11.5).

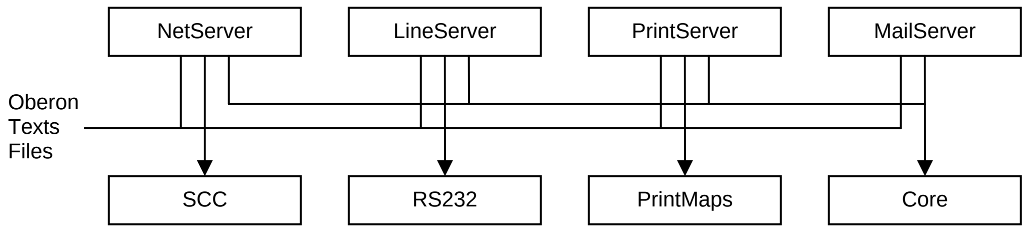


Figure 11.1 Module structure of server systems

11.2. Electronic Mail Service

The heart of an e-mail service is the set of mailboxes stored on the dedicated, central server. Each registered user owns a mailbox. The evidently necessary operations are the insertion of a message and its retrieval. In contrast to customary letter boxes, however, a retrieved message need not necessarily be removed from the box; its retrieval produces a copy. The box thereby automatically becomes a repository, and messages can be retrieved many times. This scheme calls for an additional command which removes a message from the box. Also, a command is needed for delivering a table of contents, in which presumably each message is represented by an indication of its sender and time of arrival.

The mail scheme suggested above results in the following commands:

Net.Mailbox ServerName. This command fetches a table of contents of the current user's mailbox from the specified server and displays it in a new viewer. The user's name and password must have been registered previously by the command *System.SetUser.*

Net.SendMail ServerName. The text in the marked viewer is sent to the specified server. In order to be accepted, the text must begin with at least one line beginning with "To:" and containing at least one recipient.

Net.ReceiveMail. This command is contained in the title bar (menu) of the viewer obtained when requesting the table of contents. Prior to issuing the command, the message to be read must have been specified by selecting a line in the table of contents in this viewer.

Net.DeleteMail. This command is also contained in the mailbox viewer's title bar. The message to be deleted must be selected before issuing the command.

The mail system presented here is primarily intended to serve as an exchange for short messages which are typically sent, received, read, and discarded. Mailboxes are not intended to serve as long term archives for a large and ever growing number of long pieces of text. This restrictiveness of purpose allows to choose a reasonably simple implementation and results in an efficient, practically instantaneous access to messages when the server is idle.

The Oberon mail server used at ETH also provides communication with external correspondents. It connects to an external mail server which is treated as a source and a sink for messages (almost) like other customers. Additionally, messages sent to that server need to be encoded into a standardized format, and those received need to be decoded accordingly. The parts of module *MailServer* for encoding and decoding are not described in this book. We merly divulge the fact that its design and implementation took a multiple of the time spent on the fast, local message exchange, to which we confine this presentation.

From the structures explained in Section 11.1. it follows that three agents are involved in the transfer of messages from the user into a mailbox. Therefore, additions to the server system distribute over three modules. New commands are added to module *Net* (see Section 10.4.); these procedures will be listed below. Their counterparts reside in module *NetServer* on the dedicated computer. The third agent is module *MailServer*; both are listed below in this Section. The latter handles the insertion of arriving messages into mailboxes. The path which a message traverses for insertion and retrieval is shown in Fig. 11.2. Rectangles with bold edges mark storage.

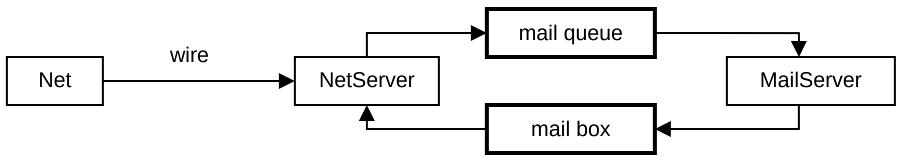


Figure 11.2 Path of messages to and from mailbox

Communication between the master station and the dedicated server runs over the network and therefore calls for an extension of its protocol (see Sect. 10.2.). The additions directly correspond to the four commands given above.

MailBox	= MDIR username password (*datastream* | *NAK* | *NPR*).
SendMail	= RML username password (*ACK datastream* | *NAK* | *NPR*).
ReceiveMail	= SML username password msgno (*datastream* | *NAK* | *NPR*).
DeleteMail	= DML username password msgno (*ACK* | *NAK* | *NPR*).

The message number is taken from the selected line in the mailbox viewer. The data transmitted are taken as (unformatted) texts. This is in contrast to file transfers, where they are taken as any sequence of bytes. The four command procedures listed below belong in module *Net*; they are listed together with the auxiliary procedures *SendText* and *ReceiveText* which closely correspond to *SendData* and *ReceiveData* (see Sect. 10.4).

We now turn our attention to the command procedures' counterparts in module *NetServer* listed in this Section. In order to explain these routines, a description of their interface with the mail server and a definition of the structure of mailboxes must precede. We begin with the simplest case, the counterpart of *SendMail*. It is the part of procedure *NetServer.Serve* which is guarded by the condition *typ = RML*, indicating a request to receive mail. As in all other services, the parameters username and password are read and the admissibility of the request is checked. The check is performed by procedure *Core.UserNo* which yields a negative number if service is to be refused. In the affirmative case, procedure *ReceiveData* obtains the message and stores it on a file, which is thereafter inserted into the mail queue as a task to be handled by the mail server at a later time. This may involve distribution of the message into several mailboxes.

Module *Core* is listed in Sect. 11.5. As mentioned before, it serves as link between the various server modules, defining the data types of the linking queues and also of mailboxes. Task queues are represented as FIFO-lists. The descriptor of type *Queue* contains a pointer to the first list element used for retrieval, and a pointer to the last element used for insertion (see Fig. 11.3). These pointers are not exported; instead, the next task is obtained by calling procedure *Core.GetTask*, and it is deleted by *Core.RemoveTask*. There e*xist two exported variables of type Queue: MailQueue consumed by MailServer, and* PrintQueue consumed by *PrintServer* (see Sect. 11.3.). (In fact, we use a third queue: *LineQueue* consumed by *LineServer*). Elements of queues are of type *TaskDesc* which specifies the file representing the data to be consumed. Additionally, it specifies the user number and identification of the task's originator. Three procedures are provided by module *Core* for handling task queues:

```
PROCEDURE InsertTask(VAR q: Queue; F: Files.File; VAR id: ARRAY OF CHAR; uno: INTEGER);

PROCEDURE GetTask(VAR q: Queue; VAR F: Files.File; VAR id: ARRAY OF CHAR; VAR uno:
INTEGER);

PROCEDURE RemoveTask(VAR q: Queue);
```

The server's counterparts of the remaining mail commands access mailboxes directly. The simplicity of the required actions - a result of a carefully chosen mailbox representation - and considerations of efficiency do not warrant a detour via task queue and mail server.

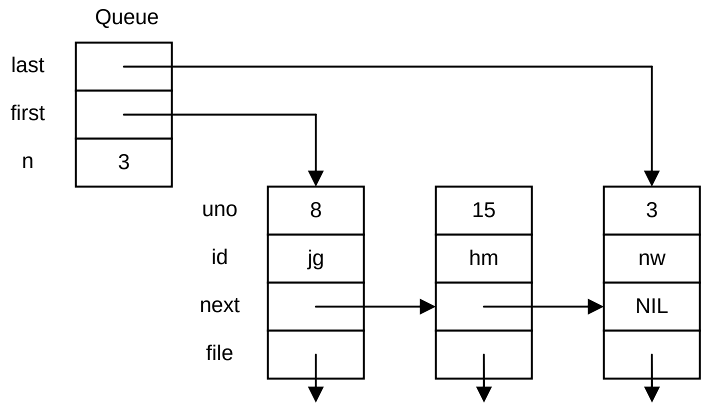


Figure 11.3 Structure of task queue

Every mailbox is represented as a file. This solution has the tremendous advantage that no special administration has to be introduced to handle a reserved partition of disk store for mail purposes. A mailbox file is partitioned into three parts: the block reservation part, the directory part, and the message part. Each part is quickly locatable, because the first two have a fixed length (32 and 31*32 = 992 bytes). The message part is regarded as a sequence of blocks (of 256 bytes), and each message occupies an integral number of adjacent blocks. Corresponding to each block, the block reservation part contains a single bit indicating whether or not the block is occupied by a message. Since the block reservation part is 32 bytes long, the message part contains at most 256 blocks, i.e. 64K bytes. The block length was chosen after an analysis of messages which revealed that the average message is less than 500 bytes long.

The directory part consists of an array of 31 elements of type *MailEntry*, a record with the following fields: *pos* and *len* indicate the index of the message's first block and the message's number of bytes; *time* and *date* indicate the message's time of insertion, and originator indicates the message's source. The entries are linked (field *next*) in chronological order of their arrival, and entry 0 serves as the list's header. It follows that a mailbox contains at most 30 messages. An example of a mailbox state is shown in Fig. 11.4.

```
MailEntry =  RECORD
               pos, next: INTEGER;
               len: LONGINT;
               time, date: INTEGER;
               originator: ARRAY 20 OF CHAR
             END ;
MResTab =    ARRAY 8 OF SET;
MailDir =    ARRAY 31 OF MailEntry;
```

We are now in a position to inspect the handler for requests for message retrieval. It is guarded by the condition *typ* = *SML*. After a validity check, the respective requestor's mailbox file is opened. The last mailbox opened is retained by the global variable MF which acts as a single entry cache. The associated user number is given by the global variable *mailuno*. Since typically several requests involving the same mailbox follow, this measure avoids the repeated reopening of the same file. Thereafter, a rider is directly positioned at the respective directory entry for reading the message's length and position in the message part. The rider is repositioned accordingly, and transmission of the message is handled by procedure *SendMail*.

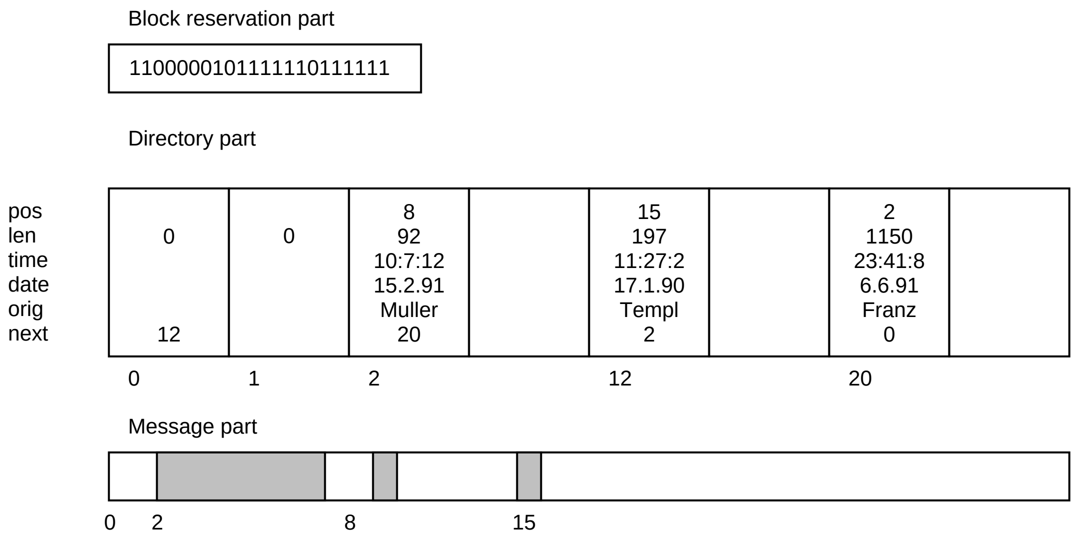


Figure 11.4 State of mailbox file

Requests for the mailbox directory are handled by the routine guarded by the condition *typ = MDIR*. The directory part must be read and converted into a text. This task is supported by various auxiliary procedures (Append) which concatenate supplied data in a buffer for latter transmission. We emphasize that this request does not require the reading of any other part of the file, and therefore is very swift.

The last of the four mail service requests (DML) deletes a specified message. Removal from the directory requires a relinking of the entries. Unused entries are marked by their *len* field having value 0. Also, the blocks occupied by the message become free. The block reservation part must be updated accordingly.

In passing we note that the use of files for representing mailboxes, in combination with the file distribution services residing on the same server station, allows anyone to access (and inspect) any mailbox. Although we do not claim that this system provides secure protection against snooping, a minimal effort for protection was undertaken by a simple encoding of messages in mailbox files. This encoding is not shown in the program listings contained in this book.

One operation remains to be explained in more detail: the processing of tasks inserted into the mail queue. It consists of the insertion of the message represented by the task's file into one or several mailboxes. It involves the interpretation of the message's header, i.e. lines containing addresses, and the construction of a new header containing the name of the originator and the date of insertion into the mailbox. These actions are performed by procedures in module *MailServer*. Its procedure *Serve* is installed as an Oberon Task, and it is guarded by the condition *Core.MailQueue.n > 0*, indicating that at least one message needs to be dispatched.

The originator's name is obtained from *Core.GetUserName(uno),* where uno is the user number obtained from the queue entry. The actual time is obtained from *Oberon.GetClock*. The form of the new header is shown by the following example:

```
From: Gutknecht
At: 12.08.91 09:34:15
```

The received message's header is then searched for recipients. Their names are listed in header lines starting with "To" (or "cc"). After a name has been read, the corresponding user number is obtained by calling *Core.UserNum*. Then the message is inserted into the designated mailbox by procedure *Dispatch*. The search for recipients continues, until a line is encountered that does not begin with "To" (or "cc"). A negative user number indicates that the given name is not registered. In this case, the message is returned to the sender, i.e. inserted into the mailbox of the sender. An exception is the recipient "all" which indicates a broadcast to all registered users.

Procedure *Dispatch* first opens the mailbox file of the user specified by the recipient number rno. If a mailbox exists, its block reservation part (mrtab) and its directory part (mdir) are read. Otherwise a new, empty box is created. Then follows the search for a free slot in the directory and, if found, the search for a sufficient number of free, adjacent blocks in the message part. The number of required blocks is given by the message length. If either no free slot exists, or there is no large enough free space for the message part, the message is returned to the sender (identified by *sno*). If also this attempt fails, the message is redirected to the postmaster (with user number 0). The postmaster is expected to inspect his mailbox sufficiently often so that no overflow occurs. If the postmaster's mailbox also overflows, the message is lost.

Only if all conditions for a successful completion are satisfied, is insertion begun. It starts with the marking of blocks in the reservation table and with the insertion of the new directory information. Table and directory are then updated on the file. Thereafter, the message with the constructed new header is written into the message part.

Perhaps it may seem to the reader that the addition of a separate module *MailServer*, together with a new Oberon Task and the machinery of the mail queue is not warranted by the relative simplicity of the insertion operation, and that it could have been incorporated into module *NetServer* just as well as message extraction. The picture changes, however, if handling of external mail is to be added, and if access to mailboxes via other channels, such as the RS-232 line, is to be provided. The presented solution is based on a modular structure that facilitates such extensions without change of existing parts. External mail routines inevitably have to cope with message formats imposed by standards. Format transformations, encoding before sending to an external server and decoding before dispatching become necessary. Indeed, these operations have inflated module *MailServer* in a surprising degree. And lastly, the queuing machinery supports the easy insertion of additional message sources and provides a welcome decoupling and relaxation of timing constraints, particularly in the case of low-speed transmission media such as telephone lines.

11.4. Miscellaneous services

There exist a few additional services that are quite desirable under the presence of a central facility, and at the same time easy to include. They are briefly described in this section.

The set of commands of the file distribution service is augmented by *Net.DeleteFiles* and *Net.Directory*, allowing the remote deletion of files and inspection of the server's directory. The command procedures are listed below and must be regarded as part of module *Net* (Sect. 10.4). They communicate with their counterparts in module *NetServer* (Sect. 11.2.) according to the following protocol:

```
DeleteFile  = DEL username password filename (ACK | NAK | NPR).
Directory   = FDIR username password prefix (datastream | NAK | NPR).
```

The directory request carries a prefix; it uses procedure *FileDir.Enumerate* to obtain all file names starting with the specified prefix. Thereby the search can be limited to the relevant section of the directory.

Since requests to the server are always guarded by a password, a facility is necessary to set and change the password stored by the server. The respective command is *Net.SetPassword,* and its handler in the server is guarded by the condition *typ* = *NPW*. The corresponding protocol is

```
NewPassword =      NPW username oldpassword
            (ACK DAT newpassword (ACK | NAK)  | NAK | NPR).
```

Finally, procedure Net.GetTime allows the workstation's real time clock to be adjusted to that of the central server. The protocol is

```
GetTime = TRQ TIM time date.
```

In concluding we summarize the entire protocol specification below. The combined server facility, comprising file distribution, electronic mail, printing, and time services is operating on a Ceres-1 computer (1 Mips) with a 2 MByte store, of which half is used by the printer's bitmap.

Summary of Protocol:

```
protocol      =  {request}.
request       =  ReceiveFile | SendFile | DeleteFile | Directory |
                 MailBox | SendMail | ReceiveMail | DeleteMail |
                 PrintStream | SendMsg | NameRequest | NewPassword | GetTime.
ReceiveFile   =  SND username password filename (datastream | NAK | NPR).
datastream    =  DAT0 data ACK1  {DATi data ACKi+1}.
SendFile      =  REC username password filename (ACK0 datastream | NAK | NPR).
datastream    =  DAT0 data ACK1  {DATi data ACKi+1}.
DeleteFile    =  DEL username password filename (ACK | NAK | NPR).
Directory     =  FDIR username password prefix (datastream | NAK | NPR).
MailBox       =  MDIR username password (datastream | NAK | NPR).
SendMail      =  RML username password (ACK datastream | NAK | NPR).
ReceiveMail   =  SML username password msgno (datastream | NAK | NPR).
DeleteMail    =  DML username password msgno (ACK | NAK | NPR).
PrintStream   =  PRT username password (ACK datastream | NAK | NPR).
SendMsg       =  MSG message ACK.
NameRequest   =  NRQ partnername [NRS].
NewPassword   =  NPW username oldpassword
                 (ACK DAT newpassword (ACK | NAK)  | NAK | NPR).
GetTime       =  TRQ TIM time date.
```

11.5. User Administration

It appears to be a universal law that centralization inevitably calls for an administration. The centralized mail and printing services make no exception. The typical duties of an administration are accounting and protection against misuse. It has to ensure that rendered services are counted and that no unauthorized user is taking advantage of the server. An additional duty is often the gathering of statistical data. In our case, accounting plays a very minor role, and the reason for the existence of the administration presented here is primarily protection.

We distinguish between two kinds of protection. The first is protection of the server's resources in general, the second is that of individual users' resources from being accessed by others. Whereas in the first case some validation of a user's identification might suffice, the second case requires the association of personal resources with user names. In any case, the central server must store data for each member of the set of registered users. Specifically, it must be able to check the admissibility of a user's request on the basis of stored information.

Evidently, a protection administration is similar in purpose and function to a lock. Quite regularly, locks are subjected to attempts of breaking them, and locksmiths are subjected to attempts of being outwitted. The race between techniques of breaking locks and that of better countermeasures is well known, and we do not even try to make a contribution to it. Our design is based on the premise that the Oberon Server operates in a harmonious environment. Nevertheless, a minimal amount of protection machinery was included. It raises the amount of effort required for breaking protection to a level which is not reached when curiosity alone is the motivation.

The data about users is held in a table in module *Core*. As was mentioned earlier, *Core* acts as connector between the various servers by means of task queues. Its second purpose is to provide the necessary access to user data via appropriate procedures.

In the simplest solution, each table entry would contain a user name only. For each request, the administration would merely test for the presence of the request's user name in the table. A significant step towards safe protection is the introduction of a password in addition to the user name. In order that a request be honoured, not only must the name be registered, but the delivered and the stored password must match. Evidently, abusive attempts would aim at recovering the stored passwords. Our solution lies in storing an encoded password. The command *System.SetUser*, which asks for a user identification and a password, immediately encodes the password, and the original is stored nowhere. The encoding algorithm is such that it is difficult to construct a corresponding decoder.

The mail service requires a third attribute in addition to identification and encoded password: the user's name as it is used for addressing messages. Identification typically consists of the user's initials; for the name we suggest the full last name of the user and discourage cryptic abbreviations.

The printing service makes an accounting facility desirable. A fourth field in each user table entry serves as a count for the number of printed pages. As a result, there are four fields: *id, name, password,* and *count*. The table is not exported, but only accessible via procedures. *Core* is a good example of a resource hiding module. The program is listed below, and a few additional comments follow here.

Procedures *UserNo(id)* and *UserNum(name)* yield the table index of the identified user; it is called *user number* and is used as a short encoding for recipients and senders within the mail server. In other servers, the number is merely used to check a request's validity.

The user information must certainly survive any intermission of server operation, be it due to software, hardware, or power failure. This requires that a copy of the user information is held on backup store (disk). The simplest solution would be to use a file for this purpose. But this would indeed make protection too vulnerable: files can be accessed easily, and we have refrained from introducing a file protection facility. Instead, the backup of the user information is held on a few permanently reserved sectors on the server machine, which are inaccessible to the file system.

Apart from procedures and variables constituting the queuing mechanism for tasks, the procedures exported from module *Core* all belong to the administration, and they can be divided into two categories. The first category contains the procedures used by the three servers presented in this Chapter, and they are *UserNo, UserNum, IncPageCount, SetPassword, GetUserName* and *GetFileName*. The second category consists of the procedures *NofUsers* and *GetUser* for inspecting table entries, and *InsertUser, DeleteUser, ClearPassword, ClearCounts*, and *Init* for making changes to the table.

The client of the latter category is a module *Users* which is needed by the human administrator of the server facility.

The reader may at this point wonder why a more advanced concept of administration has not been chosen, which would allow the human administrator to operate the server remotely. A quick analysis of the consequences of this widely used approach reveals that a substantial amount of additions to our system would be required. The issue of security and protection would become inflated into dimensions that are hardly justified for our local system. The first consequence would be a differentiation among levels of protection. The administrator would become a so-called super-user with extra priviledges, such as changing the user table. And so the game of trying to break the protection measures starts to become an interesting challenge.

We have resisted the temptation to introduce additional complexity. Instead, we assume that physical access to the server station is reserved to the administrator. Naturally, module Users and in particular the symbol file of *Core* do not belong to the public domain. In concluding, we may point out that the impossibility of activating users' programs on the server station significantly reduces the possibilities for inflicting damage from the exterior.

12 The compiler

12.1. Introduction

The compiler is the primary tool of the system builder. It therefore plays a prominent role in the Oberon System, although it is not part of the basic system. Instead, it constitutes a tool module - an application - with a single command: *Compile*. It translates program texts into machine code. Therefore, it is as a program inherently machine-dependent; it acts as the interface between source language and target computer.

In order to understand the process of compilation, the reader needs to be familiar with the source language *Oberon* defined in Appendix 1, and with the target computer *RISC,* defined in Appendix 2.

The language is defined as an infinite set of sequences of symbols taken from the language's vocabulary. It is described by a set of equations called syntax. Each equation defines a syntactic construct, or more precisely, the set of sequences of symbols belonging to that construct. It specifies how that construct is composed of other syntactic constructs. The meaning of programs is defined in terms of semantic rules governing each such construct.

Compilation of a program text proceeds by analyzing the text and thereby decomposing it recursively into its constructs according to the syntax. When a construct is identified, code is generated according to the semantic rule associated with the construct. The components of the identified construct supply parameters for the generated code.

It follows that we distinguish between two kinds of actions: analyzing steps and code generating steps. In a rough approximation we may say that the former are source language dependent and target computer independent, whereas the latter are source language independent and target computer dependent. Although reality is somewhat more complex, the module structure of this compiler clearly reflects this division. The main module of the compiler is ORP (for Oberon to RISC Parser) It is primarily dedicated to syntactic analysis, parsing. Upon recognition of a syntactic construct, an appropriate procedure is called the code generator module ORG (for Oberon to RISC Generator). Apart from parsing, ORP checks for type consistency of operands, and it computes the attributes of objects identified in declarations.

Whereas ORP mirrors the source language and is independent of a target computer, ORG reflects the target computer, but is independent of the source language.

Oberon program texts are regarded as sequences of symbols rather than sequences of characters. Symbols themselves, however, are sequences of characters. We refrain from explaining the reasons for this distinction, but mention that apart from special characters and pairs such as +, &, <=, also identifiers, numbers, and strings are classified as symbols. Furthermore, certain capital letter sequences are symbols, such as IF, END, etc. Each time the syntax analyzer (parser) proceeds to read the next symbol, it does this by calling procedure *Get*, which constitutes the so-called scanner residing in module ORS (for Oberon to RISC Scanner). It reads from the source text as many characters as are needed to recognize the next symbol.

In passing we note that the scanner alone reflects the definition of symbols in terms of characters, whereas the parser is based on the notion of symbols only. The scanner implements the abstraction of symbols. The recognition of symbols within a character sequence is called *lexical analysis*.

Ideally the recognition of any syntactic construct, say A, consisting of subconstructs, say B1, B2, ... , Bn, leads to the generation of code that depends only on (1) the semantic rules associated with A, and (2) on (attributes of) B1, B2, ... , Bn. If this condition is satisfied, the construct is said to be context-free, and if all constructs of a language are *context-free*, then also the language is context-free. Syntax and semantics of Oberon adhere to this rule, although with a significant exception. This exception is embodied by the notion of declarations. The declaration of an identifier, say *x*, attaches permanent properties to *x*, such as the fact that *x* denotes a variable and that its type is *T*. These properties are "invisible" when parsing a statement containing *x*, because the declaration of *x* is not also part of the statement. The "meaning" of identifiers is thus inherently context-dependent.

Context-dependence due to declarations is the immediate reason for the use of a global data structure which represents the declared identifiers and their properties (attributes). Since this concept stems from early assemblers where identifiers (then called *symbols*) were registered in a linear table, the term *symbol table* tends to persist for this structure, although in this compiler it is considerably more complex than an array. Basically, it grows during the processing of declarations, and it is searched while expressions and statements are processed. Procedures for building and for searching are contained in module ORB.

A complication arises from the notion of exports and imports in Oberon. Its consequence is that the declaration of an identifier x may be in a module, say M, different from where x is referenced. If x is exported, the compiler includes x together with its attributes in the *symbol file* of the compiled module M. When compiling another module which imports M, that symbol file is read and its data are incorporated into the symbol table. Procedures for reading and writing symbol files are contained in module ORB, and no other module relies on information about the structure of symbol files.

The syntax is precisely and rigorously defined by a small set of syntactic equations. As a result, the parser is a reasonably perspicuous and short program. In spite of the high degree of regularity of the target computer, the process of code generation is more complicated, as shown by module ORG.

The resulting module structure of the compiler is shown in Fig. 12.1 in a slightly simplified manner. In reality OCS is imported by all other modules due to their need for procedure *OCS.Mark*. This, however, will be explained later.

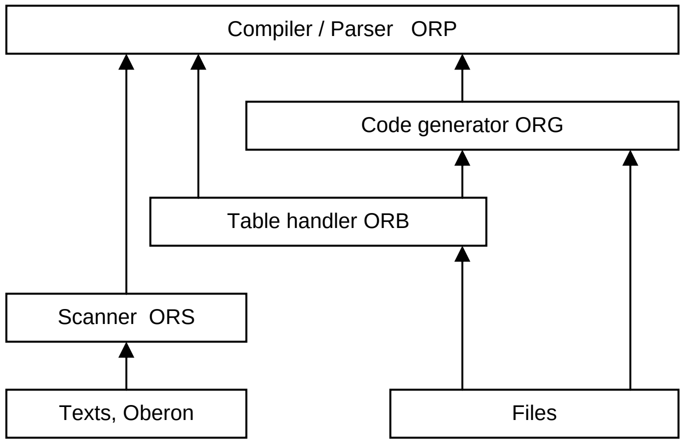


Figure 12.1 Compiler's module structure

12.2. Code patterns

Before it is possible to understand how code is generated, one needs to know *which* code is generated. In other words, we need to know the goal before we find the way leading to the goal. A fairly concise description of this goal is possible due to the structure of the language. As explained before, semantics are attached to each individual syntactic construct, independent of its context. Therefore, it suffices to list the expected code - instead of an abstract semantic rule - for each syntactic construct.

As a prerequisite to understanding the resulting instructions and in particular their parameters, we need to know where declared variables are stored, i.e. which are their addresses. This compiler uses the straight-forward scheme of sequential allocation of consecutively declared variables. An address is a pair consisting of a base address (in a register) and an offset. Global variables are allocated in the module's data section and the respective base address register is SB (Static Base, see Chapter 6). Local variables are allocated in a procedure activation record on the stack; the respective base register is SP (Stack Pointer). Offsets are positive integers.

The amount of storage needed for a variable (called its *size*) is determined by the variable's type. The sizes of basic types are prescribed by the target computer's data representation. The following holds for the RISC processor:

Type	No. of bytes
BYTE, CHAR, BOOLEAN	1
INTEGER, REAL, SET, POINTER, PROCEDURE	4

The size of an array is the size of the element type multiplied by the number of elements. The size of a record is the sum of the sizes of its fields.

A complication arises due to so-called alignment. By alignment is meant the adjustment of an address to a multiple of the variable's size. Alignment is performed for variable addresses as well as for record field offsets. The motivation for alignment is the avoidance of double memory references for variables being "distributed" over two adjacent words. Proper alignment enhances processing speed quite significantly. Variable allocation using alignment is shown by the example in Fig. 12.2.

VAR b0: BYTE; int0: INTEGER; b1: BYTE; int1: INTEGER;

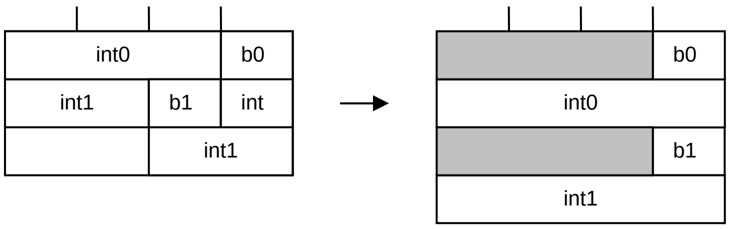


Figure 12.2. Alignment of variables

We note in passing that a reordering of the four variables lessens the number of unused bytes, as shown in Fig. 12.3.

VAR int0, int1: INTEGER; b0, b1: BYTE;

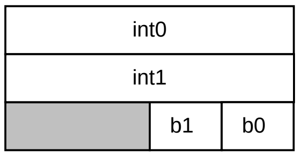


Figure 12.3. Improved order of variables

Memory instructions compute the address as the sum of a register (base) and an offset constant. Local variables use the *stack pointer* SP (R14) as base, global variables the *static base* SB (R13) Every module has its own SB value, and therefore access to global (and imported) variables requires two instructions, one for fetching the base value, and one for loading or storing data. If the compiler can determine, whether the correct base value has already been loaded into the SB register, the former instruction is omitted.

The first 7 sample patterns contain global variables only, and their base SB is assumed to hold the appropriate value. Parameters of branch instructions denote jump distances from the instruction's own location (PC-relative).

Pattern 1: *Assignment of constants.* We begin with a simple example of assigning constants to variables. The variables used in this example are global; their base register is SB. Each assignment results in a single instruction. The constant is embedded within the instruction as a literal operand.

```
MODULE Pattern1;
    VAR ch: CHAR;                    0
      k: INTEGER;                    4
      x: REAL;                       8
      s: SET;                        12
BEGIN                                             module entry code
    ch := "0";                       40000030     MOV R0 R0    30H
                                     B0D00000     STR  R0 SB     0
    k := 10;                         4000000A     MOV R0 R0    10
                                     A0D00004     STR  R0 SB     4
    x := 1.0;                        60003F80     MOV' R0 R0   3F800000H
                                     A0D00008     STR  R0 SB     8
    s := {0, 4, 8}                   40000111     MOV R0 R0    111H
                                     A0D0000C     STR  R0 SB    12
END Pattern1.                                     module exit code
```

Pattern 2: *Simple expressions:* The result of an expression containing operators is always stored in a register before it is assigned to a variable or used in another operation.

Registers for intermediate results are allocated sequentially in ascending order R0, R1, ... , R11. Integer multiplication and division by powers of 2 are represented by shifts (LSL, ASR). Similarly, the modulus by a power of 2 is obtained by masking off leading bits. The operations of set union, difference, and intersection are represented by logical operations (OR, AND).

```
MODULE Pattern2;
    VAR i, j, k, n: INTEGER;         0, 4, 8, 12
      x, y: REAL;                    16, 20
      s, t, u: SET;                  24, 28, 32
BEGIN i := (i + 1) * (i - 1);        LDR R0 SB 0
                                     ADD R0 R0 1
                                     LDR R1 SB 0
                                     SUB R1 R1 1
                                     MUL R0 R0 R1
                                     STR R0 SB 0
    k := k DIV 17;                   LDR R0 SB 8
                                     DIV R0 R0 17
                                     STR R0 SB 8
    k := 8*n;                        LDR R0 SB 12
                                     LSL R0 R0 3
                                     STR R0 SB 8
    k := n DIV 2;                    LDR R0 SB 12
                                     ASR R0 R0 1
                                     STR R0 SB 8
    k := n MOD 16;                   LDR R0 SB 12
                                     AND R0 R0 15
                                     STR r0 SB 8
    x := -y / (x - 1.0);             LDR R0 SB 16
                                     MOV' R1 R0 3F80H
                                     FSB R0 R0 R1
                                     LDR R1 SB 20
                                     FDV R0 R1 R0
                                     MOV R1 R0 0
                                     FSB R0 R1 R0
                                     STR R0 SB 16
    s := s + t * u                   LDR R0 SB 28
                                     LDR R1 SB 32
```

```
                                            AND R0 R0 R1
                                            LDR R1 SB 24
                                            OR R0 R1 R0
                                            STR R0 SB 24
END Pattern2.
```

Pattern3: *Indexed variables:* References to elements of arrays make use of the possibility to add an index value to an offset. The index must be present in a register and be multiplied by the size of the array elements. (For integers with size 4 this is done by a shift of 2 bits). Then this index is checked whether it lies within the bounds specified in the array's declaration. This is achieved by a comparison, actually a subtraction, and a subsequent branch instruction causing a trap, if the index is either negative or beyond the upper bound.

If the reference is to an element of a multi-dimensional array (matrix), its address computation involves several multiplications and additions. The address of an element $A[i_{k-1}, \dots , i_1, i_0]$ of a k-dimensional array A with lengths $n_{k-1}, \dots , n_1, n_0$ is

$$adr(A) + ((\dots ((i_{k-1} * n_{k-2}) + i_{k-2}) * n_{k-3} + \dots) * n_1 + i_1) * n_0 + i_0$$

Note that for index checks CMP is written instead of SUB to mark that the subtraction is merely a comparison, that the result remains unused and only the condition flag registers hold the result.

```
MODULE Pattern3;
    VAR i, j, k, n: INTEGER;                      0, 4, 8, 12
      a: ARRAY 10 OF INTEGER;                     16
      x: ARRAY 10, 10 OF INTEGER;                 56
      y: ARRAY 10, 10, 10 OF INTEGER;             456
BEGIN
    k := a[i];                      LDR R0 SB  0
                                    CMP R1 R0  10
                                    BLHI  R12
                                    LSL R0 R0  2
                                    ADD R0 SB R0
                                    LDR R0 R0 16
                                    STR  R0 SB  8
    n := a[5];                      LDR R0 SB  36
                                    STR  R0 SB  12
    x[i, j] := 2;                   LDR R0 SB    0
                                    CMP R1 R0 10
                                    BLHI  R12
                                    MUL R0 R0  40
                                    ADD R0 SB R0
                                    LDR R1 SB  4
                                    CMP R2 R1  10
                                    BLHI  R12
                                    LSL R1 R1 2
                                    ADD R0 R0 R1
                                    MOV R1 R0  2
                                    STR  R1 R0   56
    y[i, j, k] := 3;                LDR R0 SB  0
                                    CMP R1 R0 10
                                    BLHI  R12
                                    MUL R0 R0 400
                                    ADD R0 SB R0
                                    LDR R1 SB  4
                                    CMP R2 R1 10
                                    BLHI  R12
                                    MUL R1 R1 40
                                    ADD R0 R0 R1
                                    LDR R1 SB  8
                                    CMP R2 R1 10
```

```
                                                  BLHI  R12
                                                  LSL R1 R1 2
                                                  ADD R0 R0 R1
                                                  MOV R1 R0  3
                                                  STR  R1 R0  456
    y[3, 4, 5] := 6                               MOV R0 R0   6
                                                  STR  R0 SB 1836
END Pattern3.
```

Pattern 4: *Record fields and pointers:* Fields of records are accessed by computing the sum of the record's (base) address and the field's offset. If the record variable is statically declared, the sum is computed by the compiler.

```
MODULE Pattern4;
    TYPE Ptr = POINTER TO Node;
      Node = RECORD num: INTEGER;             0
        name: ARRAY 8 OF CHAR;                4
        next: Ptr                             12
      END ;
    VAR p, q: Ptr;                            12, 16
      r: Node;                                20

BEGIN
  r.num := 10;                    MOV R0 R0  10
                                  STR  R0 SB  20
  p.num := 6                      LDR R0 SB  12  (p)
                                  MOV R1 R0   6
                                  STR  R1 R0   0
  p.name[7] := "0";               LDR R0 SB  12
                                  MOV R1 R0  30H
                                  STR  R1 R0  11   (4+7)
  p.next := q;                    LDR R0 SB   12
                                  LDR R1 SB   16
                                  STR  R1 R0   12
  p.next.next := NIL              LDR R0 SB   12  (p)
                                  LDR R0 R0    12  (p.next)
                                  MOV R1 R0  0   (NIL)
                                  STR  R1 R0  12   (p.next.next)
END Pattern4.
```

Pattern 5: *Boolean expressions, If statements:* Conditional statements imply that parts of them are skipped. This is done by the use of branch instructions whose operand specifies the distance of the branch. The instructions refer to the condition-register as an implicit operand. Its value is determined by a preceding instruction, typically a compare or a bit-test instruction.

The Boolean operators & and OR are purposely not defined as total functions, but rather by the equations

```
p & q      =  if p then q else FALSE
p OR q     =  if p then TRUE else q
```

Consequently, Boolean operators must be translated into branches too. Evidently, branches stemming from if statements and branches stemming from Boolean operators should be merged, if possible. The resulting code therefore does not necessarily mirror the structure of the if statement directly, as can be seen from the code in *Pattern5*. We must conclude that code generation for Boolean expressions differs in some aspects from that for arithmetic expressions.

The example of *Pattern5* is also used to exhibit the code resulting from the standard procedures INC, DEC, INCL, and EXCL. These procedures provide an opportunity to use shorter code in those cases where a single two-operand instruction suffices, i.e. when one of the arguments is identical with the destination.

```
MODULE Pattern5;
   VAR n: INTEGER; s: SET;                0, 4
BEGIN
   IF n = 0 THEN                          LDR R0 SB  0
                                          CMP R0 R0  0
                                          BNE  3
         INC(n)                           LDR R0 SB  0
                                          ADD R0 R0  1
                                          STR R0 SB  0
   END ;
   IF (n >= 0) & (n < 100) THEN           LDR SB R0 ...
                                          LDR R0 SB  0   (n)
                                          CMP R0 R0  0
                                          BLT  6
                                          LDR R0 SB  0
                                          CMP R0 R0  100
                                          BGE  3
         DEC(n)                           LDR R0 SB  0
                                          SUB R0 R0  1
                                          STR R0 R0  0
   END ;
   IF ODD(n) OR (n IN s) THEN             LDR SB R0 ...
                                          LDR R0 SB  0   (n)
                                          AND R0 R0  1
                                          BNE  5
                                          LDR R0 SB  4   (s)
                                          LDR R1 SB  0
                                          ADD R1 R1  1
                                          ROR R0 R0 R1
                                          BPL  2
         n := -1000                       MOV R0 R0 -1000
                                          STR R0 SB  0
   END ;
   IF n < 0 THEN                          LDR SB R0  ...
                                          LDR R0 SB  0
                                          CMP R0 R0  0
                                          BGE  3
         s := {}                          MOV R0 R0  0   {}
                                          STR R0 SB  4
                                          B  17
   ELSIF n < 10 THEN                      LDR SB R0 ...
                                          LDR R0 SB  0
                                          CMP R0 R0  10
                                          BGE  3
         s := {0}                         MOV R0 R0  1
                                          STR          R0 SB  4
                                          B  10
   ELSIF n < 100 THEN                     LDR SB R0 ...
                                          LDR R0 SB  0
                                          CMP R0 R0 100
                                          BGE  3
         s := {1}                         MOV R0 R0  2
                                          STR R0 SB  4
                                          B  3
   ELSE
         s := {2}                         MOV R0 R0 4
                                          LDR SB R0 ...
                                          STR R0 SB  4
   END
END Pattern5.
```

Pattern 6: *While and repeat statements.*

```
MODULE Pattern6;
  VAR i: INTEGER;
BEGIN i := 0;                                MOV R0 R0      0
                                             STR  R0 SB      0
  WHILE i < 10 DO                            LDR SB R0      ...
                                             LDR R0 SB      0
                                             CMP R0 R0  10
                                             BGE  4
      i := i + 2                             LDR R0 SB  0
                                             ADD R0 R0  2
                                             STR R0 SB  0
  END ;                                      B  -8
  REPEAT i := i - 1                          LDR SB R0      ...
                                             LDR R0 SB      0
                                             SUB R0 R0      1
                                             STR  R0 SB      0
  UNTIL i = 0                                LDR R0 SB      0
                                             CMP R0 R0  0
                                             BNE  -7
END Pattern6.
```

Pattern 7: *For statements.*

```
MODULE Pattern7;
  VAR i, m, n: INTEGER;
BEGIN
  FOR i := 0 TO n-1 DO                       MOV R0 R0   0
                                             LDR R1 SB   8
                                             SUB R1 R1 1
                                             CMP LNK R0 R1
                                             BGT    7
                                             STR  R0 SB   0
    m := 2*m                                 LDR R0 SB   4
                                             LSL R0 R0  1
                                             STR  R0 SB  4
  END                                        LDR R0 SB   0
                                             ADD R0 R0  1
                                             B   -11
END Pattern7.
```

Pattern 8: *Proper procedures:* Procedure bodies are surrounded by a prolog (entry code) and an epilog (exit code). They reposition the stack pointer SP (see Chapter 6), which holds the address of the procedure activation record on the stack. The immediate value of the first instruction indicates the space taken by variables local to the procedure, rounded up to the next multiple of 4.

Procedure calls use a branch and link (BL) instruction. Parameters are loaded into registers prior to the call and pushed on the stack after the call. Every parameter occupies a multiple of 4 bytes. In the case of value parameters the value is loaded, and in the case of VAR-parameters, the variable's address is loaded.

```
MODULE Pattern8;
  VAR i: INTEGER;

  PROCEDURE P(x: INTEGER; VAR y: INTEGER);
    VAR z: INTEGER;
  BEGIN                               SUB SP SP     16           adjust SP
                                      STR  LNK SP   0            push ret adr
                                      STR  R0 SP     4           push x
                                      STR  R1 SP     8           push @y
    z := x;                           LDR R0 SP      4           x
                                      STR  R0 SP     12          z
    y := z                            LDR R0 SP      12          z
```

```
                                    LDR R1 SP      8           @y
                                    STR  R0 R1     0           y
  END P;                            LDR LNK SP     0           pop ret adr
                                    ADD SP SP     16
                                    B R15

BEGIN P(5, i)                       MOV R0 R0      5
                                    ADD R1 SB      0           @i
                                    BL     -14                 call
END Pattern8.
```

Pattern 9: *Function procedures.* They are handled in exactly the same manner as proper procedures, except that a result is returned in register R0. If the function is called in an expression at a place where intermediate results are held in registers, these values are put onto the stack before the call, and they are restored after return (not shown here).

```
MODULE Pattern9;
  VAR x: REAL;

  PROCEDURE F(x: REAL): REAL;
  BEGIN                             SUB SP SP 8
                                    STR LNK SP 0               push ret adr
                                    STR R0 SP  4               push x
    IF x >= 1.0 THEN                LDR R0 SP  4
                                    MOV' R1 R0 3F80H
                                    FSB R0 R0 R1
                                    BLT       4
      x := F(F(x))                  LDR R0 SP  4
                                    BL -9
                                    BL -10
                                    STR R0 SP 4
    END ;
    RETURN x                        LDR R0 SP 4
 END F;                             LDR LNK SP 0               pop ret adr
                                    ADD SP SP 8
                                    B R15
END Pattern9.
```

Pattern 10: *Dynamic array parameters* are passed by loading a descriptor on the stack, regardless of whether they are value- or VAR- parameters. The descriptor consists of the actual variable's address and the array's length. (Only one-dimensional dynamic arrays are handled).

Elements of dynamic arrays are accessed like those of static arrays. However, even when the index is a constant, the check cannot be performed by the compiler.

```
MODULE Pattern10;
  VAR a: ARRAY 12 OF INTEGER;

  PROCEDURE P(x: ARRAY OF INTEGER);
      VAR i, n: INTEGER;
  BEGIN                             SUB SP SP  20
                                    STR LNK SP  0
                                    STR R0 SP 4                x
                                    STR R1 SP 8  x.len
      n := x[i];                    LDR R0 SP 12               i
                                    LDR R1 SP  8               x.len
                                    CMP R2 R0 R1
                                    BLHI R12
                                    LSL R0 R0  2
                                    LDR R1 SP 4                x
```

```
                                     ADD R0 R1 R0
                                     LDR R0 R0       0
                                     STR  R0 SP  16
      x[i+1] := n+5                  LDR R0 SP  12             i
                                     ADD R0 R0  1
                                     LDR R1 SP  8              x.len
                                     CMP R2 R0 R1
                                     BLHI  R12
                                     LSL R0 R0  2
                                     LDR R1 SP  4              x
                                     ADD R0 R1 R0
                                     LDR R1 SP  16             n
                                     ADD R1 R1 5
                                     STR  R1 R0  0
    END P;                           LDR LNK SP      0
                                     ADD SP SP     20
                                     B  R15

  BEGIN P(a);                        ADD R0 SB  0              a
                                     MOV R1 R0 12              a.len
                                     BL  -29
  END Pattern10.
```

Pattern 11: *Sets.* This code pattern exhibits the construction of sets. If the specified elements are constants, the set value is computed by the compiler. Otherwise, sequences of move and shift instructions are used. Since shift instructions do not check whether the shift count is within sensible bounds, the results are unpredictable, if elements outside the range 0 .. 31 are involved.

```
  MODULE Pattern11;
    VAR s: SET; m, n: INTEGER;
  BEGIN
    s := {m};                        LDR R0 SB  4              m
                                     MOV R1 R0  1
                                     LSL R0 R1 R0
                                     STR  R0 SB  0             s
    s := {0 .. n};                   LDR R0 SB   8             n
                                     MOV R1 R0  -2
                                     LSL R0 R1 R0
                                     XOR R0 R0  -1
                                     STR  R0 SB   0
    s := {m .. 31};                  LDR R0 SB    4            m
                                     MOV R1 R0  31
                                     MOV R2 R0   -2
                                     LSL R1 R2 R1
                                     MOV R2 R0     -1
                                     LSL R0 R2 R0
                                     XOR R0 R0 R1
                                     STR  R0 SB  0             s
    s := {m .. n};                   LDR R0 SB   4             m
                                     LDR R1 SB   8             n
                                     MOV R2 R0  -2
                                     LSL R1 R2 R1
                                     MOV R2 R0     -1
                                     LSL R0 R2 R0
                                     XOR R0 R0 R1
                                     STR  R0 SB  0             s
    IF n IN {2, 3, 5, 7, 11, 13} THEN MOV R0 R0  28ACH
                                     LDR R1 SB    8
                                     ADD R1 R1   1
                                     ASR' R0 R0 R1
                                     BPL   2
      m := 1                          MOV R0 R0   1
                                     STR  R0 SB   4            m
    END
  END Pattern11.
```

Pattern 12: *Imported variables and procedures:* When a procedure is imported from another module, its address is unavailable to the compiler. Instead, the procedure is identified by a number obtained from the imported module's *symbol file*. In place of the offset, the branch instruction holds (1) the number of the imported module, (2) the number of the imported procedure, and (3) a link in the list of BL instructions calling an external procedure. This list is traversed by the linking loader, that computes the actual offset (fixup, see Chapter 6).

Imported variables are also referenced by a variable's number. In general, an access required two instructions. The first loads the static base register SB from a global table with the address of that module's data section. The module number of the imported variable serves as index. The second instruction loads the address of the variable, using the actual offset fixed up by the loader.

In the following example, modules Pattern12a and Pattern12b both export a procedure and a variable. They are referenced from the importing module Pattern12c.

```
MODULE Pattern12a;
  VAR k*: INTEGER;

  PROCEDURE P*;
  BEGIN k := 1
  END P;

END Pattern12a.


MODULE Pattern12b;
    VAR x*: REAL;

    PROCEDURE Q*;
    BEGIN x := 1
    END Q;

END Pattern12b.

MODULE Pattern12c;
    IMPORT Pattern12a, Pattern12b;

    VAR i: INTEGER; y: REAL;
BEGIN
    i := Pattern12a.k;         8D10xxxx     LDR SB 1 link     Pattern12a
                               80D00000     LDR R0 SB 0       Pattern12a.k
                               8D00xxxx     LDR SB 0 link     Pattern12c
                               A0D00000     STR R0 SB 0       Pattern12c.i
    y := Pattern12b.x;         8D20xxxx     LDR SB 2 link     Pattern12b
                               80D00000     LDR R0 SB 0       Pattern12b.x
                               8D00xxxx     LDR SB 0 link     Pattern12c
                               A0D00004     STR  R0 SB 4      Pattern12c.y
END Pattern12c.
```

Pattern 13: *Record extensions with pointers:* Fields of a record type R1, which is declared as an extension of a type R0, are simply appended to the fields of R0, i.e. their offsets are greater than those of the fields of R0. When a record is statically declared, its type is known by the compiler. If the record is referenced via a pointer, however, this is not the case. A pointer bound to a base type R0 may well refer to a record of an extension R1 of R0. Type tests (and type guards) allow to test for the actual type. This requires that a type can be identified at the time of program execution. Because the language defines name equivalence instead of structural equivalence of types, a type may be identified by a number. We use the address of a unique type descriptor for this purpose.

Therefore, type tests consist of a simple address comparison which is very fast. Type descriptors are stored in the module's area for data. Their address is called *type tag*. The tag of a (dynamically allocated) variable is stored as a prefix to its record (with offset -8).

A type descriptor contains - in addition to information stored for use by the garbage collector - a table of tags of all its base types. If, for instance, a type R2 is an extension of R1 which is an extension of R0, the descriptor of R2 contains the tags of R1 and R0 as shown in Fig. 12.4. The table has a fixed number of 3 entries.

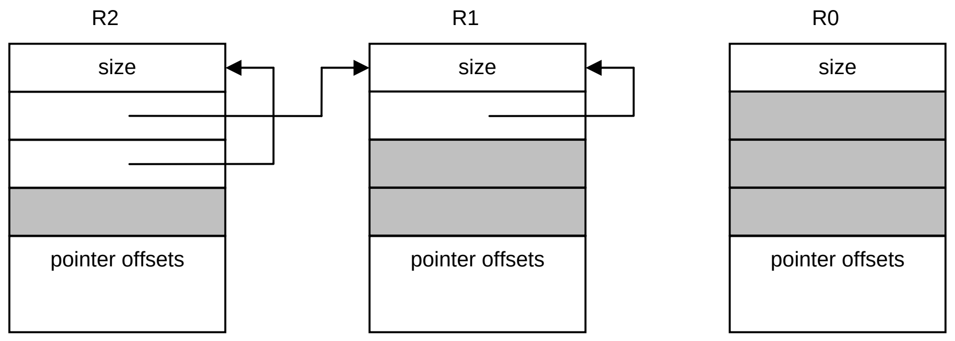


Figure 12.4 Type descriptors

A type test of the form *p IS T* then, consists of a comparison of the type tag of p^ at address p-8 with the tag held in the descriptor of T at the extension level of the type of p^. A type guard *p(T)* is synonymous to the statement

IF ~(p IS T) THEN abort END

The following example features 3 record types with associated pointer types, and hence also 3 type descriptors. Each descriptor is 5 words long. Their addresses, and therefore their tags, are 0, 20, and 40 respectively.

```
 0 00000020 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
20 00000020 00014006 FFFFFFFF FFFFFFFF FFFFFFFF
40 00000020 00014005 00028001 FFFFFFFF FFFFFFFF
```

```
MODULE Pattern13;
  TYPE
    P0 = POINTER TO R0;
    P1 = POINTER TO R1;
    P2 = POINTER TO R2;
    R0 = RECORD x: INTEGER END ;
    R1 = RECORD (R0) y: INTEGER END ;
    R2 = RECORD (R1) z: INTEGER END ;
  VAR
    p0: P0;                   60
    p1: P1;                   64
    p2: P2;                   68
BEGIN
  p0.x := 0;                  LDR R0 SB 60
                              MOV R1 R0 0        p0.x
                              STR  R1 R0 0       no type check
  p1.y := 1;                  LDR R0 SB 64
                              MOV R1 R0 1
                              STR  R1 R0 4       p1.y
  p0(P1).y := 3;              LDR R0 SB 60       p0
                              LDR R1 R0 -8       tag(p0)
                              LDR R1 R1 4
                              ADD R2 SB 20       TD P1
                              CMP R3 R2 R1
```

```
                                BLNE  R12
                                MOV R1 R0 3
                                STR  R1 R0 4          p0.z
    p0(P2).z := 5;              LDR R0 SB 60          p0
                                LDR R1 R0 -8          tag(p0)
                                LDR R1 R1 8
                                ADD R2 SB 40          TD P2
                                CMP R3 R2 R1
                                BLNE  R12
                                MOV R1 R0 5
                                STR  R1 R0 8          p0.z
    IF p1 IS P2 THEN            LDR R0 SB 64          p1
                                LDR R1 R0 -8          tag(p1)
                                LDR R1 R1 8
                                ADD R2 SB 40          TD P2
                                CMP R3 R2 R1
                                BNE 2
      p0 := p2                  LDR R0 SB     68
                                STR R0 SB     60
    END
  END Pattern13.
```

Pattern 14: *Record extensions as VAR parameters:* Records occurring as VAR-parameters may also require a type test at program execution time. This is because VAR-parameters effectively constitute hidden pointers. Type tests and type guards on VAR-parameters are handled in the same way as for variables referenced via pointers, with a slight difference, however. Statically declared record variables may be used as actual parameters, and they are not prefixed by a type tag. Therefore, the tag has to be supplied together with the variable's address when the procedure is called, i.e. when the actual parameter is established. Record structured VAR-parameters therefore consist of address and type tag. This is similar to dynamic array descriptors consisting of address and length.

```
 0 00000020 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
20 00000020 00014006 FFFFFFFF FFFFFFFF FFFFFFFF
```

```
MODULE Pattern14;
    TYPE
      R0 = RECORD a, b, c: INTEGER END ;
      R1 = RECORD (R0) d, e: INTEGER END ;
    VAR
      r0: R0;                   40
      r1: R1;                   52

    PROCEDURE P(VAR r: R0);
    BEGIN                       ...

      r.a := 1;                 LDR R1 SP 4           r
                                STR R0 R1 0           r.a
      r(R1).d := 2              LDR R0 SP 8           tag(r)
                                LDR R0 R0 4
                                ADD R1 SB 20          R1
                                CMP R2 R1 R0
                                BLNE  R12
                                MOV R0 R0 2
                                LDR R1 SP 4           r
                                STR R0 R1 12          r.d
    END P;                      ...

BEGIN                           ...
    P(r0);                      ADD R0 SB 40          r0
                                ADD R1 SB 0           tag(R0)
                                BL  P
    P(r1)                 ADD R0 SB 52          r1
                          ADD R1 SB 20          tag(R1)
                          BL  P
END Pattern14.            ...
```

Pattern 15: *Array assignments and strings.*

```
MODULE Pattern15;
  VAR s0, s1: ARRAY 32 OF CHAR;

  PROCEDURE P(x: ARRAY OF CHAR);
  END P;

BEGIN s0 := "ABCDEF";     ADD R0 SB     0       @s0
                          ADD R1 SB    64       @"ABCDEF"
                          LDR R2 R1     0
                          ADD R1 R1     4
                          STR R2 R0     0
                          ADD R0 R0     4
                          ASR R2 R2    24       test for 0X
                          BNE       -6

  s0 := s1;               ADD R0 SB     0       @s0
                          ADD R1 SB    32       @s1
                          MOV R2 R0     8       len
                          LDR R3 R1     0
                          ADD R1 R1     4
                          STR R3 R0     0
                          ADD R0 R0     4
                          SUB R2 R2     1
                          BNE       -6

  P(s1);                  ADD R0 SB    32       @s1
                          MOV R1 R0    32       len
                          BL    -38             P

  P("012345");            ADD R0 SB    72       @"012345"
                          MOV R1 R0     7       len  (incl 0X)
                          BL    -42             P

  P("%")                  ADD R0 SB    80       @"%"
                          MOV R1 R0     2       len
                          BL    -46             P
END Pattern15.
```

Pattern 16: *Predeclared procedures.*

```
MODULE Pattern16;
  VAR m, n: INTEGER;
      x: REAL; u: SET;
      a, b: ARRAY 10 OF INTEGER;
      s, t: ARRAY 16 OF CHAR;
BEGIN
  INC(m);                         ADD R0 SB    0         @m
                                  LDR R1 R0   0
                                  ADD R1 R1  1
                                  STR R1 R0   0
  DEC(n, 10);                     ADD R0 SB  4           @n
                                  LDR R1 R0   0
                                  SUB R1 R1  10
                                  STR R1 R0   0
  INCL(u, 3);                     ADD R0 SB  12          @u
                                  LDR R1 R0   0
                                  OR R1 R1   8           {3}
                                  STR R1 R0   0
  EXCL(u, 7);                 ADD R0 SB  12          @u
                                  LDR R1 R0   0
                                  AND R1 R1  -129        -{7}
                                  STR R1 R0   0
  ASSERT(m < n);              LDR R0 SB  0
                                  LDR R1 SB  4
                                  CMP R0 R0 R1
                                  BLGE  R12
  UNPK(x, n);                 LDR R0 SB  8           x
                                  ASR R1 R0   23
                                  SUB R1 R1  127
                                  STR R1 SB   4          n
                                  LSL R1 R1  23
                                  SUB R0 R0 R1
                                  STR R0 SB   8          x
  PACK(x, n);                 LDR R0 SB   8          x
                                  LDR R1 SB   4          n
                                  LSL R1 R1  23
                                  ADD R0 R0 R1
                                  STR R0 SB   8          x
  s := "0123456789";          ADD R0 SB  96          @s
                                  ADD R1 SB  128         adr of string
                                  LDB R2 R1   0          loop
                                  ADD R1 R1  4
                                  STB R2 R0   0
                                  ADD R0 R0   4
                                  ASR R2 R2  24
                                  BNE      -6
  IF s < t THEN               ADD R0 SB  96          @s
                                  ADD R1 SB  112         @t
                                  LDB R2 R0   0          loop
                                  ADD R0 R0   1
                                  LDB R3 R1   0
                                  ADD R1 R1   1
                                  CMP R4 R2 R3
                                  BNE   2
                                  CMP R4 R2  0
                                  BNE   -8
                                  BGE   3

      m := 1                  MOV R0 R0  1
                                  STR R0 SB   0          m

  END
END Pattern16.
```

Pattern 17: *Predeclared functions.*

```
MODULE Pattern17;
  VAR m, n: INTEGER;
      x, y: REAL;
      b: BOOLEAN; ch: CHAR;
BEGIN
  n := ABS(m);                LDR R0 SB 0            m
                              CMP R0 R0 0
                              BGE 2
                              MOV R1 R0 0
                              SUB R0 R1 R0
                              STR R0 SB 4            n
  y := ABS(x);                LDR R0 SB 8            x
                              LSL R0 R0 1
```

```
                                ROR R0 R0 1
                                STR R0 SB 12          y
   b := ODD(n);                 LDR R0 SB 4           n
                                AND R0 R0 1
                                BEQ  2
                                MOV R0 R0 1
                                B  1
                                MOV R0 R0 0
                                STB R0 SB 16          b
   n := ORD(ch);                LDB R0 SB 17          ch
                                STR R0 SB 4           n
   n := FLOOR(x);               LDR R0 SB 8           x
                                MOV' R1 R0 4B00H
                                FAD" R0 R0 R1         floor
                                STR R0 SB 4           n
   y := FLT(m);                 LDR R0 SB 0           m
                                MOV' R1 R0 4B00H
                                FAD' R0 R0 R1         float
                                STR R0 SB 12          y
   n := LSL(m, 3);              LDR R0 SB 0           m
                                LSL R0 R0 3
                                STR R0 SB 4           n
   n := ASR(m, 8);              LDR R0 SB 0
                                ASR R0 R0 8
                                STR R0 SB 4
   m := ROR(m, n);              LDR R0 SB 0
                                LDR R1 SB 4
                                ROR R0 R0 R1
                                STR R0 SB 0
 END Pattern17.
```

12.3. Internal data structures and module interfaces

12.3.1. Data structures

In Section 12.1 it was explained that declarations inherently constitute context-dependence of the translation process. Although parsing still proceeds on the basis of a context-free syntax and relies on contextual information only in a few isolated instances, information provided by declarations affects the generated code significantly. During the processing of declarations, their information is transferred into the "symbol table", a data structure of considerable complexity, from where it is retrieved for the generation of code.

This dynamic data structure is defined in module ORB in terms of two record types called *Object* and *Struct*. These types pervade all other modules with the exception of the scanner. They are therefore explained before further details of the compiler are discussed (see module ORB below).

For each declared identifier an instance of type *Object* is generated. The record holds the identifier and the properties associated with the identifier given in its declaration. Since Oberon is a *statically typed* language, every object has a type. It is represented in the record by its *typ* field, which is a pointer to a record of type *Struct*. Since many objects may be of the same type, it is appropriate to record the type's attributes only once and to refer to them via a pointer. The properties of type *Struct* will be discussed below.

The kind of object which a table entry represents is indicated by the field *class*. Its values are denoted by declared integer constants: *Var* indicates that the entry describes a variable, *Con* a constant, *Fld* a record field, *Par* a VAR-parameter, and *Proc* a procedure. Different kinds of entries carry different attributes. A variable or a parameter carries an address, a constant has a value, a record field has an offset, and a procedure has an entry address, a list of parameters, and a result type. For each class the introduction of an extended record type would seem advisable. This was not done, however, for three reasons. First, the compiler was first formulated in (a subset of)

Modula-2 which does not feature type extension. Second, not making use of type extensions would make it simpler to translate the compiler into other languages for porting the language to other computers. And third, all extensions were known at the time the compiler was planned. Hence extensibility provided no argument for the introduction of a considerable variety of types. The simplest solution lies in using the multi-purpose fields *val* and *dsc* for class-specific attributes. For example, *val* holds an address for variables, parameters, and procedures, an offset for record fields, and a value for constants.

The definition of a type yields a record of type *Struct*, regardless of whether it occurs within a type declaration, in which case also a record of type *Object* (class = *Typ*) is generated, or in a variable declaration, in which case the type remains anonymous. All types are characterized by a form and a size. A type is either a basic type or a constructed type. In the latter case it refers to one or more other types. Constructed types are arrays, records, pointers, and procedural types. The attribute *form* refers to this classification. Its value is an integer

Just as different object classes are characterized by different attributes, different forms have different attributes. Again, the introduction of extensions of type *Struct* was avoided. Instead, some of the fields of type *Struct* remain unused in some cases, such as for basic types, and others are used for form-specific attributes. For example, the attribute *base* refers to the element type in the case of an array, to the result type in the case of a procedural type, to the type to which a pointer is bound, or to the base type of a (extended) record type. The attribute *dsc* refers to the parameter list in the case of a procedural type, or to the list of fields in the case of a record type.

As an example, consider the following declarations. The corresponding data structure is shown in Fig. 12.5. For details, the reader is referred to the program listing of module ORB and the respective explanations.

```
CONST  N = 100;
TYPE   Ptr = POINTER TO Rec;
       Rec = RECORD n: INTEGER; p, q: Ptr END ;
VAR    k: INTEGER;
       a: ARRAY N OF INTEGER;
PROCEDURE P(x: INTEGER): INTEGER;
```

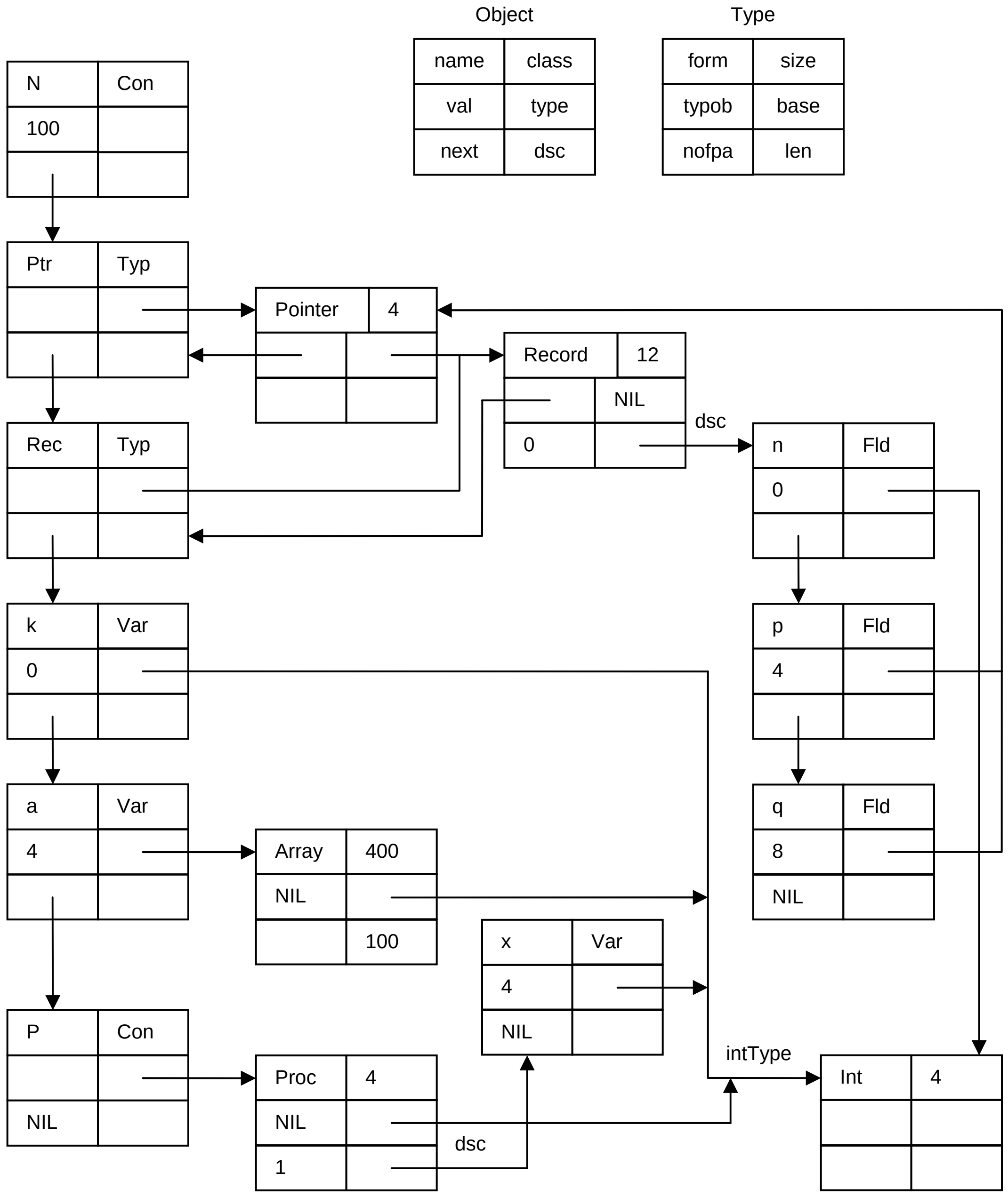


Figure 12.5. Representation of declarations

Only entries representing constructed types are generated during compilation. An entry for each basic type is established by the compiler's initialization. It consists of an *Object* holding the standard type's identifier and a *Struct* indicating its form, denoted by one of the values *Byte, Bool, Char, Int, Real,* or *Set*. The object records of the basic types are anchored in global pointer variables in module ORB (which actually should be regarded as constants).

Not only are entries created upon initialization for basic types, but also for all standard procedures. Therefore, every compilation starts out with a symbol table reflecting all standard, pervasive identifiers and the objects they stand for.

We now return to the subject of *Objects*. Whereas objects of basic classes (*Const, Var, Par, Fld, Typ, SProc, SFunc* and *Mod*) directly reflect declared identifiers and constitute the context in which statements and expressions are compiled, compilations of expressions typically generate anonymous entities of additional, non-basic modes. Such entities reflect selectors, factors, terms, etc., i.e. constituents of expressions and statements. As such, they are of a transitory nature and hence are not represented by records allocated on the heap. Instead, they are represented by record variables local to the processing procedures and are therefore allocated on the stack. Their type is called *Item* and is a slight variation of the type Object. Items are not referenced via pointers.

Let us assume, for instance, that a term *x*y* is parsed. This implies that the operator and both factors have been parsed already. The factors *x* and *y* are represented by two variables of type *Item* of *Var* mode. The resulting term is again described by an item, and since the product is transitory, i.e. has significance only within the expression of which the term is a constituent, it is to be held in a temporary location, in a register. In order to express that an item is located in a register, a new, non-basic mode *Reg* is introduced.

Effectively, all non-basic modes reflect the target computer's architecture, in particular its addressing modes. The more addressing modes a computer offers, the more item modes are needed to represent them. The additional item modes required by the RISC processor are. They are declared in module ORG:

Reg	direct register mode
RegI	indirect register mode
Cond	condition code mode

The use of the types *Object, Item,* and *Struct* for the various modes and forms, and the meaning of their attributes are explained in the following tables:

	Objects:		Items:		
	class	val	a	b	r
0	Undef				
1	Const	val	val		
2	Var	adr	adr	base	
3	Par	adr	adr	off	
4	Fld	off	off		
5	Typ	TDadr	TDadr	modno	
6	SProc	num			
7	SFunc	num			
8	Mod				
10	Reg				regno
11	RegI		off		regno
12	Cond		Tjmp	Fjmp	condition code

Structures:

	form	nofpar	len	dsc	base
7	Pointer				base type
10	ProcTyp	nofpar		param	result type
12	Array		nofel		element typ
13	Record	ext lev	desc adr	fields	extension type

Items have an attribute called *lev* which is part of the address of the item. Positive values denote the level of nesting of the procedure in which the item is declared; lev = 0 implies a global object. Negative values indicate that the object is imported from the module with number *-lev*.
The three types *Object, Item,* and *Struct* are defined in module ORB, which also contains procedures for accessing the symbol table.

12.3.2. Module interfaces

Before embarking on a presentation of the compiler's main module, the parser, an overview of its remaining modules is given in the form of their interfaces. The reader is invited to refer to them when studying the parser.

The interface of the scanner module ORS is simple. It defines the numeric values of all symbols. But its chief constituent is procedure *Get*. Each call yields the next symbol from the source text, identified by an integer. Global variables represent attributes of the read symbol in certain cases. If a number was read, *ival* or *rval* hold its numeric value. If an identifier or a string was read, *str* holds the ASCII values of the characters read.

Procedure *Mark* serves to generate a diagnostic output indicating a brief diagnostic and the scanner's current position in the source text. This procedure is located in the scanner, because only the scanner has access to its current position. *Mark* is called from all other modules.

```
DEFINITION ORS;  (*Scanner*)
  IMPORT Texts, Oberon;

  TYPE Ident = ARRAY 32 OF CHAR;
  VAR ival, slen: INTEGER;
      rval: REAL;
      id: Ident;
      str: ARRAY 256 OF CHAR;
      errcnt: BOOLEAN;

  PROCEDURE Mark (msg: ARRAY OF CHAR);
  PROCEDURE Get (VAR sym: INTEGER);
  PROCEDURE Init (source: Texts.Text; pos: INTEGER);
END ORS.
```

Module ORB defines the basic data structures representing declared objects and their types. It also contains procedures for accessing these structures. *NewObj* serves to insert a new identifier, and it returns a pointer to the allocated object. *ThisObj* returns the pointer to the object whose name equals the global scanner variable *ORS.id. Thisimport* and *thisfield* deliver imported objects and record fields with names equal to *ORS.id.*

Procedure *Import* serves to read the specified symbol file and to enter its identifier in the symbol table (class = *Mod*). Finally, *Export* generates the symbol file of the compiled module, containing descriptions of all objects and structures marked for export.

```
DEFINITION ORB;  (*Base table handler*)
  TYPE
    Object = POINTER TO ObjDesc;
    Type = POINTER TO TypeDesc;
    ObjDesc = RECORD
          class, lev, exnp: INTEGER;
          expo, rdo: BOOLEAN;
          next, dsc: Object;
          type: Type;
          name: ORS.Ident;
          val: INTEGER
    END ;
    TypeDesc = RECORD
          form, ref, mno: INTEGER;  (*ref is used for import/export only*)
          nofpar: INTEGER;  (*for records: extension level*)
          len: INTEGER;  (*for records: address of descriptor*)
          dsc, typobj: Object;
          base: Type;
          size: INTEGER
    END ;

  VAR topScope: Object;
    byteType, boolType, charType, intType, realType, setType,
    nilType, noType, strType: Type;
  PROCEDURE Init;
  PROCEDURE Close;
  PROCEDURE NewObj (VAR obj: Object; id: ORS.Ident; class: INTEGER);
  PROCEDURE thisObj (): Object;
  PROCEDURE thisimport (mod: Object): Object;
  PROCEDURE thisfield (rec: Type): Object;
  PROCEDURE OpenScope;
  PROCEDURE CloseScope;
  PROCEDURE Import (VAR modid, modid1: ORS.Ident);
  PROCEDURE Export (VAR modid: ORS.Ident;
          VAR newSF: BOOLEAN; VAR key: INTEGER);
END ORB.
```

Module ORG contains the procedures for code generation. The names of these procedures indicate the respective constructs for which code is to be produced. Note that an individual code generator procedure is provided for every standard, predefined procedure. This is necessary, because they generate in-line code.

```
  DEFINITION ORG;
    CONST WordSize* = 4;
    TYPE Item* = RECORD
        mode*: INTEGER;
        type*: ORB.Type;
        a*, b*, r: INTEGER;
        rdo*: BOOLEAN  (*read only*)
      END ;
    VAR pc: INTEGER;

    PROCEDURE MakeConstItem*(VAR x: Item; typ: ORB.Type; val: INTEGER);
    PROCEDURE MakeRealItem*(VAR x: Item; val: REAL);
    PROCEDURE MakeStringItem*(VAR x: Item; len: INTEGER);
    PROCEDURE MakeItem*(VAR x: Item; y: ORB.Object; curlev: INTEGER);
    PROCEDURE Field*(VAR x: Item; y: ORB.Object);   (* x := x.y *)
    PROCEDURE Index*(VAR x, y: Item);   (* x := x[y] *)
    PROCEDURE DeRef*(VAR x: Item);
    PROCEDURE BuildTD*(T: ORB.Type; VAR dc: INTEGER);
    PROCEDURE TypeTest*(VAR x: Item; T: ORB.Type; varpar, isguard: BOOLEAN);

    PROCEDURE Not*(VAR x: Item);   (* x := ~x,  Boolean operators *)
    PROCEDURE And1*(VAR x: Item);   (* x := x & *)
    PROCEDURE And2*(VAR x, y: Item);
    PROCEDURE Or1*(VAR x: Item);   (* x := x OR *)
    PROCEDURE Or2*(VAR x, y: Item);

    PROCEDURE Neg*(VAR x: Item);   (* x := -x, arithmetic operators *)
    PROCEDURE AddOp*(op: LONGINT; VAR x, y: Item);   (* x := x +- y *)
    PROCEDURE MulOp*(VAR x, y: Item);   (* x := x * y *)
    PROCEDURE DivOp*(op: INTEGER; VAR x, y: Item);   (* x := x op y *)
    PROCEDURE RealOp*(op: INTEGER; VAR x, y: Item);   (* x := x op y *)

    PROCEDURE Singleton*(VAR x: Item);  (* x := {x}, set operators *)
    PROCEDURE Set*(VAR x, y: Item);   (* x := {x .. y} *)
    PROCEDURE In*(VAR x, y: Item);  (* x := x IN y *)
    PROCEDURE SetOp*(op: INTEGER; VAR x, y: Item);   (* x := x op y *)

    PROCEDURE IntRelation*(op: INTEGER; VAR x, y: Item);   (* x := x < y *)
    PROCEDURE SetRelation*(op: INTEGER; VAR x, y: Item);   (* x := x < y *)
    PROCEDURE RealRelation*(op: INTEGER; VAR x, y: Item);   (* x := x < y *)
    PROCEDURE StringRelation*(op: INTEGER; VAR x, y: Item);   (* x := x < y *)

    PROCEDURE StrToChar*(VAR x: Item);   (*assinments*)
    PROCEDURE Store*(VAR x, y: Item); (* x := y *)
    PROCEDURE StoreStruct*(VAR x, y: Item); (* x := y *)
    PROCEDURE CopyString*(VAR x, y: Item);  (*from x to y*)
```

```
   PROCEDURE VarParam*(VAR x: Item; ftype: ORB.Type);  (*parameters*)
   PROCEDURE ValueParam*(VAR x: Item);
   PROCEDURE OpenArrayParam*(VAR x: Item);
   PROCEDURE StringParam*(VAR x: Item);

   PROCEDURE For0*(VAR x, y: Item);   (*For Statements*)
   PROCEDURE For1*(VAR x, y, z, w: Item; VAR L: LONGINT);
   PROCEDURE For2*(VAR x, y, w: Item);

   (* Branches, procedure calls, procedure prolog and epilog *)
   PROCEDURE Here*(): LONGINT;
   PROCEDURE FJump*(VAR L: LONGINT);
   PROCEDURE CFJump*(VAR x: Item);
   PROCEDURE BJump*(L: LONGINT);
   PROCEDURE CBJump*(VAR x: Item; L: LONGINT);
   PROCEDURE Fixup*(VAR x: Item);
   PROCEDURE PrepCall*(VAR x: Item; VAR r: LONGINT);
   PROCEDURE Call*(VAR x: Item; r: LONGINT);
   PROCEDURE Enter*(parblksize, locblksize: LONGINT; int: BOOLEAN);
   PROCEDURE Return*(form: INTEGER; VAR x: Item; size: LONGINT; int: BOOLEAN);

   (* In-line code procedures*)
   PROCEDURE Increment*(upordown: LONGINT; VAR x, y: Item);
   PROCEDURE Include*(inorex: LONGINT; VAR x, y: Item);
   PROCEDURE Assert*(VAR x: Item);
   PROCEDURE New*(VAR x: Item);
   PROCEDURE Pack*(VAR x, y: Item);
   PROCEDURE Unpk*(VAR x, y: Item);
   PROCEDURE Led*(VAR x: Item);
   PROCEDURE Get*(VAR x, y: Item);
   PROCEDURE Put*(VAR x, y: Item);
   PROCEDURE Copy*(VAR x, y, z: Item);
   PROCEDURE LDPSR*(VAR x: Item);
   PROCEDURE LDREG*(VAR x, y: Item);

   (*In-line code functions*)
   PROCEDURE Abs*(VAR x: Item);
   PROCEDURE Odd*(VAR x: Item);
   PROCEDURE Floor*(VAR x: Item);
   PROCEDURE Float*(VAR x: Item);
   PROCEDURE Ord*(VAR x: Item);
   PROCEDURE Len*(VAR x: Item);
   PROCEDURE Shift*(fct: LONGINT; VAR x, y: Item);
   PROCEDURE ADC*(VAR x, y: Item);
   PROCEDURE SBC*(VAR x, y: Item);
   PROCEDURE UML*(VAR x, y: Item);
   PROCEDURE Bit*(VAR x, y: Item);
   PROCEDURE Register*(VAR x: Item);
   PROCEDURE H*(VAR x: Item);
   PROCEDURE Adr*(VAR x: Item);
   PROCEDURE Condition*(VAR x: Item);

   PROCEDURE Open*(v: INTEGER);
   PROCEDURE SetDataSize*(dc: LONGINT);
   PROCEDURE Header*;
   PROCEDURE Close*(VAR modid: ORS.Ident; key, nofent: LONGINT);
 END ORG.
```

12. 4. The Parser

The main module ORP constitutes the parser. Its single command *Compile* - at the end of the program listing - identifies the source text according to the Oberon command conventions. It then calls procedure *Module* with the identified source text as parameter. The command forms are:

ORP.Compile @	The most recent selection identifies the beginning of the source text.
ORP.Compile ^	The most recent selection identifies the name of the source file.
ORP.Compile f0 f1 ... ~	f0, f1, ... are the names of source files.

File names and the characters @ and ^ may be followed by an option specification /s. Option s enables the compiler to overwrite an existing symbol file, thereby invalidating clients.

The parser is designed according to the proven method of top-down, recursive descent parsing with a look-ahead of a single symbol. The last symbol read is represented by the global variable *sym*. Syntactic entities are mirrored by procedures of the same name. Their goal is to recognize the specified construct in the source text. The start symbol and corresponding procedure is *Module*. The principal parser procedures are shown in Fig. 12.6., which also exhibits their calling hierarchy. Loops in the diagram indicate recursion in the syntactic definition.

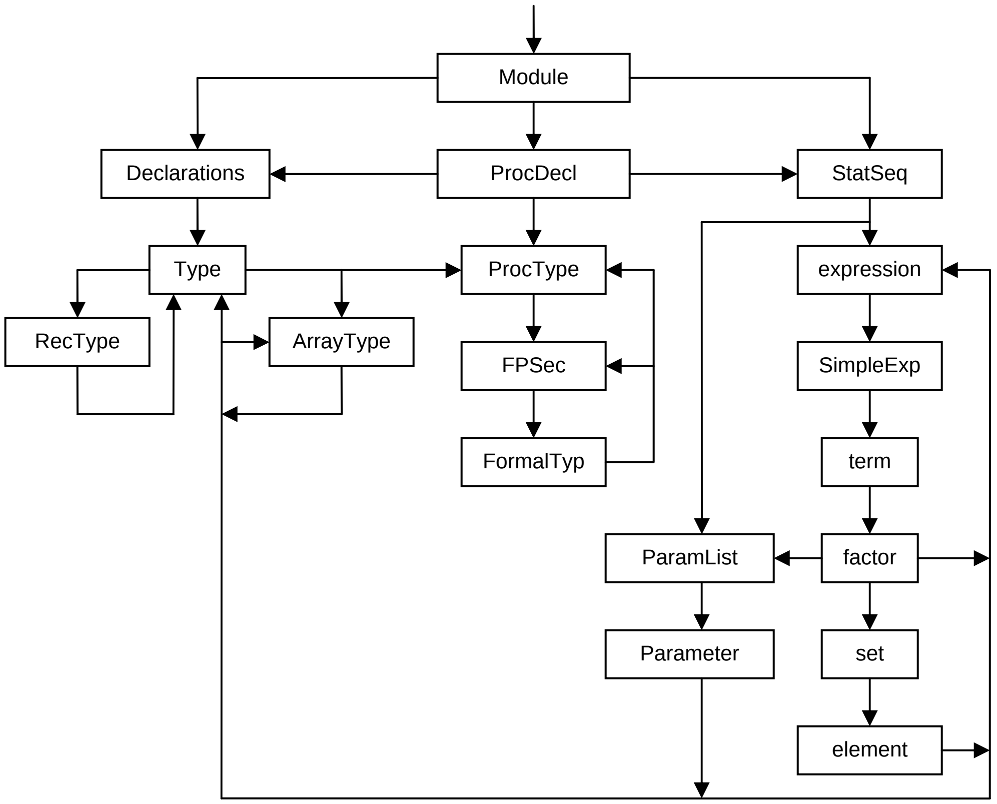


Figure 12.6 Parser procedure hierarchy

The rule of parsing strictly based on a single-symbol look-ahead and without reference to context is violated in three places. The prominent violation occurs in statements. If the first symbol of a statement is an identifier, the decision of whether an assignment or a procedure call is to be recognized is based on contextual information, namely the class of the identified object. The second violation occurs in *qualident*; if the identifier x preceding a period denotes a module, it is recognized together with the subsequent identifier as a qualified identifier. Otherwise x supposedly denotes a record variable. The third violation is made in procedure *selector*; if an identifier is followed by a left parenthesis, the decision of whether a procedure call or a type guard is to be recognized is again made on the basis of contextual information, namely the mode of the identified object.

A fairly large part of the program is devoted to the discovery of errors. Not only should they be properly diagnosed. A much more difficult requirement is that the parsing process should continue on the basis of a good guess about the structure that the text should most likely have. The parsing process must continue with some assumption and possibly after skipping a short piece of the source text. Hence, this aspect of the parser is mostly based on heuristics. Incorrect assumptions about the nature of a syntactic error lead to secondary error diagnostics. There is no way to avoid them. A reasonably good result is obtained by the fact that procedure *ORS.Mark* inhibits an error report, if it lies less than 10 characters ahead of the last one. Also, the language Oberon is designed with the property that most large constructs begin with a unique symbol, such as IF, WHILE, CASE, RECORD, etc. These symbols facilitate the recovery of the parsing process in the erroneous text. More problematic are open constructs which neither begin nor end with key symbols, such as types, factors, and expressions. Relying on heuristics, the source text is skipped up to the first occurrence of a symbol which may begin a construct that follows the one being parsed. The employed scheme may not be the best possible, but it yields quite acceptable results and keeps the amount of program devoted to the handling of erroneous texts within justifiable bounds.

Besides the parsing of text, the Parser also performs the checking for type consistency of objects. This is based on type information held in the global table, gained during the processing of declarations, which is also handled by the routines which parse. Thereby an unjustifiably large number of very short procedures is avoided. However, the strict target-computer independence of the parser is violated slightly: Information about variable allocation strategy including alignment, and about the sizes of basic types is used in the parser module. Whereas the former violation is harmless, because the allocation strategy is hardly controversial, the latter case constitutes a genuine target-dependence embodied in a number of explicitly declared constants. Mostly these constants are contained in the respective type definitions, represented by records of type *Type* initialized by ORB. The following procedures allocate objects and generate elements of the symbol table:

Declarations	Object(Con), Object(Typ), Object(Var)
ProcedureDeclaration	Object(xProc)
FormalType	Object(Var), Object(Par)
ORB.Import	Object(Mod)
RecordType	Object(Fld), Type(Record)
ArrayType	Type(Array)
ProcedureType	Type(ProcTyp)
Type	Type(Pointer)
FormalType	Type(Array)

An inherently nasty subject is the treatment of forward references in a single-pass compiler. In Oberon, there are two such cases:

1. Forward declarations of procedures. They have been eliminated from the revision of the Oberon language in the year 2007 as they should be avoided if ever possible. If it is impossible, a remedy is to declare a variable of the given procedure type, and assign the procedure to be forwarded to this variable. The nastiness of procedure forward declarations originates in the necessity to specify parameters and result type in the forward declaration. These must be repeated in the actual procedure declaration, and one expects that a compiler verifies the equality (or equivalence) of the two declarations. This is a heavy burden for a case that very rarely occurs.

2. Forward declarations of pointer types also constitutes a nasty exception, but its exclusion would be difficult to justify. If in a pointer declaration the base type (to which the pointer is bound) is not found in the symbol table, a forward reference is therefore automatically assumed. An entry for the pointer type is generated anyway (see procedure *Type*) and an element is inserted in the list of pointer base types to be fixed up. This list is headed by the global variable *pbsList.* When later in the text a declaration of a record type is encountered with the same identifier, the forward entry is recognized and the proper link is established (see procedure *Declarations*).

The compiler must check for undefined forward references when the current declaration scope is closed. This check is performed at the end of procedure *Declarations.*

The with statement had been eliminated from the language in its revision of 2007. Here it reappears in the form of a case statement, whose cases are not labelled by integers, but rather by types. What formerly was written as

```
IF x IS T1 THEN
    WITH x: T1 DO ... x ... END
ELSIF x IS T2 THEN
    WITH x: T2 DO ... x ... END
ELSIF ...
END
```

is now written more simply and more efficiently as

```
CASE x OF
 T1: ... x ... |
 T2: ... x ... |
...
END
```

where *T1* and *T2* are extensions of the type *T0* of the case variable *x*. Compilation of this form of case statement merges the regional type guard of the former with statements with the type test of the former if statements. This case statement represents the only case where a symbol table entry - the type of *x* - is modified during compilation of statements. When the end of the with statement is reached, the change must be reverted.

12.5. The scanner

The scanner module ORS embodies the lexicographic definitions of the language, i.e. the definition of abstract symbols in terms of characters. The scanner's substance is procedure *Get*, which scans the source text and, for each call, identifies the next symbol and yields the corresponding integer code. It is most important that this process be as efficient as possible. Procedure *Get* recognizes letters indicating the presence of an identifier (or reserved word), and digits signalling the presence of a number. Also, the scanner recognizes comments and skips them. The global variable *ch* stands for the last character read.

A sequence of letters and digits may either denote an identifier or a key word. In order to determine which is the case, a search is made in a table containing all key words for each would-be identifier. This table is sorted alphabetically and according to the length of reserved words. It is initialized when the compiler is loaded.

The presence of a digit signals a number. Procedure *Number* first scans the subsequent digits (and letters) and stores them in a buffer. This is necessary, because hexadecimal numbers are denoted by the postfix letter H (rather than a prefix character). The postfix letter X specifies that the digits denote a character.

There exists one case in the language Oberon, where a look-ahead of a single character does not suffice to identify the next symbol. When a sequence of digits is followed by a period, this period may either be the decimal point of a real number, or it may be the first element of a range symbol (..). Fortunately, the problem can be solved locally as follows: If, after reading digits and a period, a second period is present, the number symbol is returned, and the look-ahead variable *ch* is assigned the special value 7FX. A subsequent call of *Get* then delivers the range symbol. Otherwise the period after the digit sequence belongs to the (real) number.

12.6. Searching the symbol table, and handling symbol files

12.6.1. The structure of the symbol table

The symbol table constitutes the context in which statements and expressions are parsed. Each procedure establishes a scope of visibility of local identifiers. The records registering identifiers belonging to a scope are linked as a linear list. They are of type *Object*. Each object has a type.

Types are represented by records of type *Type*. These two types pervade the entire compiler, and they are defined as follows:

```
TYPE Object = POINTER TO ObjDesc;
  Type = POINTER TO TypeDesc;

  ObjDesc = RECORD
      class, lev, exno: INTEGER;
      expo, rdo: BOOLEAN;   (*exported / read-only*)
      next, dsc: Object;
      type: Type;
      name: ORS.Ident;
      val:  INTEGER
    END ;

  TypeDesc = RECORD
      form, ref, mno: INTEGER;  (*ref is only used for import/export*)
      nofpar: INTEGER;  (*for procedures; extension level for records*)
      len:  INTEGER;  (*for arrays, len < 0 => open array; for records: adr of descriptor*)
      dsc, typobj: Object;
      base: Type;  (*for arrays, records, pointers*)
      size:  INTEGER;  (*in bytes; always multiple of 4, except for Byte, Bool and Char*)
    END ;
```

Procedures for generating and searching the lists are contained in module ORB. If a new identifier is to be added, procedure *NewObj* first searches the list, and if the identifier is already present, a double definition is diagnosed. Otherwise the new element is appended, thereby preserving the order given by the source text.

Procedures, and therefore also scopes, may be nested. Each scope is represented by the list of its declared identifiers, and the list of the currently visible scopes are again connected as a list. Procedure *OpenScope* appends an element and procedure *CloseScope* removes it. The list of scopes is anchored in the global variable *topScope* and linked by the field *dsc.* It is treated like a stack. It consists of elements of type *Object*, each one being the header (*class = Head*) of the list of declared entities. As an example, the procedure for searching an object (with name *ORS.id*) is shown here:

```
PROCEDURE thisObj*(): Object;
  VAR s, x: Object;
BEGIN s := topScope;
  REPEAT x := s.next;
    WHILE (x # NIL) & (x.name # ORS.id) DO x := x.next END ;
    s := s.dsc
  UNTIL (x # NIL) OR (s = NIL);
  RETURN x
END thisObj;
```

A snapshot of a symbol table for an example with nested scopes is shown in Fig. 12.6. It is taken when the following declarations are parsed and when the statement *S* is reached.

```
VAR x: INTEGER;

PROCEDURE P(u: INTEGER);
BEGIN ... END P;

PROCEDURE Q(v: INTEGER);
  PROCEDURE R(w: INTEGER);
  BEGIN S END R;
BEGIN ... END Q;
```

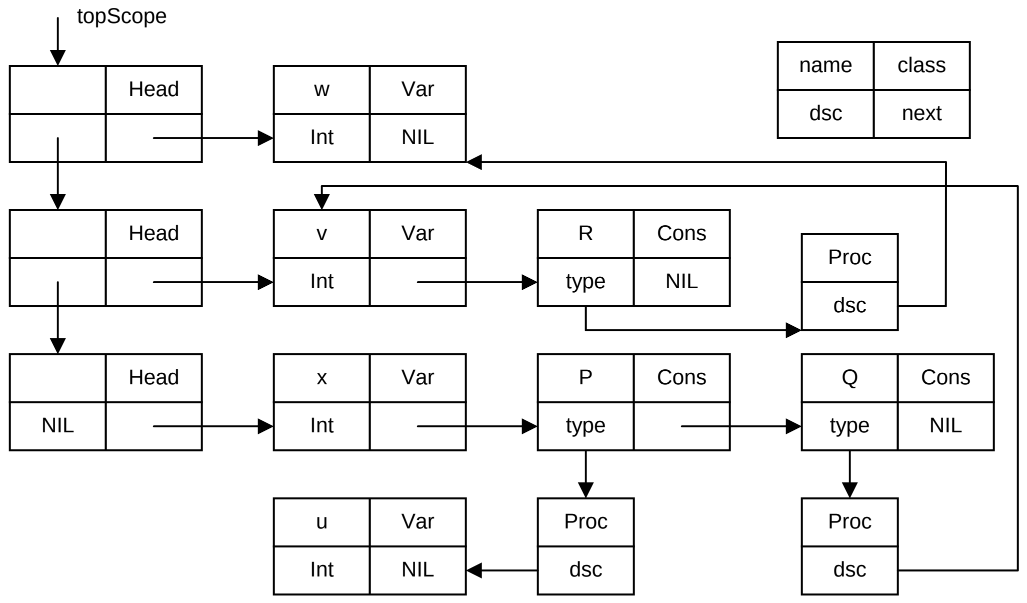


Fig. 12.7 Snapshot of a symbol table

A search of an identifier proceeds first through the scope list, and for each header its list of object records is scanned. This mirrors the scope rule of the language and guarantees that if several entities carry the same identifier, the most local one is selected. The linear list of objects represents the simplest implementation by far. A tree structure would in many cases be more efficient for searching, and would therefore seem more recommendable. Experiments have shown, however, that the gain in speed is marginal. The reason is that the lists are typically quite short. The superiority of a tree structure becomes manifest only when a large number of global objects is declared. We emphasize that when a tree structure is used for each scope, the linear lists must still be present, because the order of declarations is sometimes relevant in interpretation, e.g. in parameter lists.

Not only procedures, but also record types establish their own local scope. The list of record fields is anchored in the type record's field *dsc*, and it is searched by procedure *thisField*. If a record type *R1* is an extension of *R0*, then *R1*'s field list contains only the fields of the extension proper. The base type *R0* is referenced by the *BaseTyp* field of *R1*. Hence, a search for a field may have to proceed through the field lists of an entire sequence of record base types.

12.6.2. Symbol files

The major part of module ORB is devoted to input and output of symbol files. A symbol file is a linearized form of an excerpt of the symbol table containing descriptions of all exported (marked) objects. All exports are declared in the global scope. Procedure *Export* traverses the list of global objects and outputs them to the symbol file.

The structure of a symbol file is defined by the syntax specified below. The following terminal symbols are class and form specifiers or reference numbers for basic types with fixed values:

Classes: Con = 1, Var = 2, Par = 3, Fld = 4; Typ = 5

Forms: Byte = 1, Bool = 2, Char = 3, Int = 4, LInt = 5, Set = 6,
Pointer = 7, NoTyp = 9, ProcTyp = 10, Array = 12, Record = 13

Syntax:

```
    SymFile  = null key name versionkey {object}.
    object  =    (CON name type (value | exno) | TYP name type [{fix} 0] | VAR name type expno).
    type  =      ref (PTR type | ARR type len | REC type {field} 0 | PRO type {param} 0].
    field  =    FLD name type offset.
    param  =    (VAR | PAR) type.
```

A procedure type description is contains a parameter list. Similarly, a record type description with form specifier *Record* contains the list of field descriptions. Note that a procedure is considered as a constant of a procedure type.

Objects have types, and types are referenced by pointers. These cannot be written on a file. The straight-forward solution would be to use the type identifiers as they appear in the program to denote types. However, this would be rather crude and inefficient, and second, there are anonymous types, for which artificial identifiers would have to be generated.

An elegant solution lies in consecutively numbering types. Whenever a type is encountered the first time, it is assigned a unique *reference number.* For this purpose, records (in the compiler) of type *Type* contain the field *ref*. Following the number, a description of the type is then written to the symbol file. When the type is encountered again during the traversal of the data structure, only the reference number is issued, with negative sign. The global variable *ORB.Ref* functions as the running reference number.

When reading a symbol file, a positive reference number is followed by the type's description. A pointer to the type read is assigned to the global table *typtab* with the reference number as index. When a negative reference number is read (it is *not* followed by a type description), then the type is identified by *typtab[-ref]* (see procedure *InType*). In the following example, types are identified by their reference number (e.g. R #14), and later referenced by this number (^14).

```
MODULE A;
  CONST Ten* = 10; Dollar* = "$";
  TYPE R* = RECORD u*: INTEGER; v*: SET END ;
    S* = RECORD w*: ARRAY 4 OF R END ;
    P* = POINTER TO R;
    A* = ARRAY 8 OF INTEGER;
    B* = ARRAY 4, 5 OF REAL;
    C* = ARRAY 10 OF S;
    D* = ARRAY OF CHAR;
  VAR x*: INTEGER;
  PROCEDURE Q0*;
  BEGIN END Q0;
  PROCEDURE Q1*(x, y: INTEGER): INTEGER;
  BEGIN RETURN x+y END Q1;
END A.
```

```
class = CON Ten [^4]   10
class = CON Dollar [^3]   36
class = TYP R [#14  form = REC [^9]  exno = 1  extlev = 0  size = 8 { v [^6]   4 u [^4]   0}]()
class = TYP S [#15  form = REC [^9]  exno = 2  extlev = 0  size = 32 { w [#0  form = ARR [^14]  len = 4  size = 32]   0}]()
class = TYP P [#16  form = PTR [^14]]()
class = TYP A [#17  form = ARR [^4]  len = 8  size = 32]()
class = TYP B [#18  form = ARR [#0  form = ARR [^5]  len = 5  size = 20]  len = 4  size = 80]()
class = TYP C [#19  form = ARR [^15]  len = 10  size = 320]()
class = TYP D [#20  form = ARR [^3]  len = -1  size = 8]()
class = VAR x [^4]    3
class = CON Q0 [#0  form = PRO [^9]()]    4
class = CON Q1 [#0  form = PRO [^4]( class = VAR [^4] class = VAR [^4])]    5
```

After a symbol file has been generated, it is compared with the file from a previous compilation of the same module, if one exists. Only if the two files differ and if the compiler's s-option is enabled, is the old file replaced by the new version. The comparison is made by comparing byte after byte without consideration of the file's structure. This somewhat crude approach was chosen because of its simplicity and yielded good results in practice.

A symbol file must not contain addresses (of variables or procedures). If they did, most changes in the program would result in a change of the symbol file. This must be avoided, because changes in the implementation (rather than the interface) of a module are supposed to remain invisible to the clients. Only changes in the interface are allowed to effect changes in the symbol file, requiring recompilation of all clients. Therefore, addresses are replaced by *export numbers*. The variable *exno* (global in *ORP*) serves as running number (see *ORP.Declarations* and *ORP.ProcedureDecl*). The translation from export number to address is performed by the loader. Every code file contains a list (table) of addresses (of variables and entry points for procedures). The export number serves as index in this table to obtain the requested address. Export numbers are generated by the parser.

Objects exported from some module *M1* may refer in their declaration to some other module *M0* imported by *M1*. It would be unacceptable, if an import of *M1* would then also require the import of *M0*, i.e. imply the automatic reading of *M01*'s symbol file. It would trigger a chain reaction of imports that must be avoided. Fortunately, such a chain reaction can be avoided by making symbol files *self-contained,* i.e. by including in every symbol file the description of entities that stem from other modules. Such entities are always types.

The inclusion of types imported from other modules seems simple enough to handle: type descriptions must include a reference to the module from which the type was imported. This reference is the name and key of the respective module. However, there exists one additional complication that cannot be ignored. Consider a module *M1* importing a variable *x* from a module *M0*. Let the type *T* of *x* be defined in module *M0*. Also, assume *M1* to contain a variable y of type *M0.T*. Evidently, *x* and *y* are of the same type, and the compiler compiling *M2* must recognize this fact. Hence, when importing *M0* during compilation of *M1*, the imported element *T* must not only be registered in the symbol table, but it must also be recognized as being identical to the *T* already imported from *M2* directly. It is rather fortunate that the language definition specifies equivalence of types on the basis of names rather than structure, because it allows type tests at execution time to be implemented by a simple address comparison.

The measures to be taken to satisfy the new requirements are as follows:

1. Every type element in a symbol file is given a module number. Before a type description is emitted to the file.

2. If a type to be exported has a name and stems from another, imported module, then also the name and key of the module are attached, from which the type stems (see end of procedure *ORB.OutType* and end of *ORB.InType*).

An additional detail is worth being mentioned here: Hidden pointers. We recall that individual fields of exported record types may be hidden. If marked (by an asterisk) they are exported and therefore visible in importing modules. If not marked, they are not exported and remain invisible, and evidently seem to be omissible in symbol files. However, this is a fallacy. They need to be included in symbol files, although without name, because of meta information to be provided for garbage collection. This is elucidated as follows:

Assume that a module M1 declares a global pointer variables of a type imported from module M0.

```
MODULE M0;
  TYPE Ptr = POINTER TO Rec0;
    Rec0* = RECORD p*, q: Ptr ... END ;
END M0.

MODULE M1;
  VAR p: M0.Ptr;
    r: RECORD f: M0.Ptr; ... END ;
END M1.
```

Here *p* and *r.f* are roots of data structures that must be visited by the garbage collector. If they are not, they will not be marked, and therefore collected with disastrous and entirely unpredictable consequences. The crux is that not only exported pointers (p.p) must be listed, but also hidden ones (p.q), although they are not accessible in module M1.

We chose to include hidden pointers in symbol files without their names, but with their type being of the form *ORB.NilTyp*. This must be considered in procedure ORG.FindPtrs, where the condition *typ.form = ORB.Pointer* must be extended to *(typ.form = ORB.Pointer) OR (typ.form = ORB.NilTyp).*

But the story does not end here. Assume that in the example above module M1 declares a type *Rec1* as a n extension of *M0.Rec0*. This requires the generation of a type descriptor. And this descriptor must include not only field *p*, but also the hidden field *q*. This is achieved by also extending the condition *typ.form = ORB.Pointer* in *ORG.FindPtrFlds* to *(typ.form = ORB.Pointer) OR (typ.form = ORB.NilTyp).*

12. 7. The code generator

The routines for generating instructions are contained in a single module: ORG. They are fairly numerous, and therefore the interface of ORG is quite large. It is a procedural interface. This implies that there is no "intermediate code" or "intermediate data structure" between parser and code generator. This is one reason for the compactness of the code generator. The other is the regularity and simplicity of the processor architecture. In order to understand the following material, the reader is supposed to be familiar with this architecture (Appendix 2) and the generated code patterns for individual language constructs (Section 12.2).

A distinguishing feature of this compiler is that parsing proceeds top-down according to the principle of recursive descent in the parsing tree. This implies that for every syntactic construct a specific procedure is called. It carries the same name as the construct. It also implies that properties of the parsed construct can be represented by parameters of the parsing procedures. Consider, for example, the construct of simple expression:

```
SimpleExpression  =  term {"+" term}.
```

The corresponding parsing procedure is

```
PROCEDURE SimpleExpression(VAR x: Item);
  VAR y: Item;
BEGIN term(x);
  WHILE sym = plus DO ORS.Get(sym); term(y); ORG.AddOp(x, y) END
END SimpleExpression
```

The generating procedure *AddOp* receives two parameters representing the operands, and returns the result through the first parameter. This scheme carries the invaluable advantage of using operands efficiently allocated on the stack rather than dynamically allocated on the heap and subject to automatic storage retrieval (garbage collection). Here the processed operands quietly disappear from the stack upon exit from the parser procedure.

The parameters representing syntactic constructs are of type *Item* defined in ORG. This data type is rather similar to the type *Object* (in ORB). After all, it serves the same purpose; but it represents internal items rather than declared objects.

```
TYPE Item = RECORD
    mode: INTEGER;
    type: ORB.Type;
    a, b, r: INTEGER;
    rdo: BOOLEAN  (*read only*)
END
```

The attribute *class* of *Object* is renamed *mode* in *Item*. In fact, in some sense different classes evoke different (corresponding) *addressing modes* as featured by the processor architecture. According to the architecture, additional modes may have to be introduced. Thanks to the simplicity of RISC, only three are needed:

Reg = 10;	The item *x* is located in register *x.r*
RegI = 11;	The item *x* is addressed indirectly through register *x.r* plus offset *x.a*
Cond = 12;	The item is represented by the condition bit registers

Instructions are emitted sequentially and emitted by the four procedures *Put0, Put1, Put2, Put3*. They directly correspond to the instruction formats of the RISC processor (see Chapter 11). The instructions are stored in the array *code* and the compiler variable *pc* serves as running index.

```
PROCEDURE Put0(op, a, b, c: INTEGER);          format F0
PROCEDURE Put1(op, a, b, im: INTEGER);         format F1
PROCEDURE Put2(op, a, b, off: INTEGER);        format F2
PROCEDURE Put3(op, cond, off: INTEGER);        format F3
```

12.7.1. Expressions

Expressions consist of operands and operators. They are evaluated and have a value. First, a number of make-procedures transform objects into items (see Section 12.3.2). The principal one is *MakeItem*. Typical objects are variables (class, mode = *Var*). Global variables are addressed with base register SB (x.r = 13), local variables with the stack pointer SP (x.r = 14). VAR-parameters are addressed indirectly; the address is on the stack (class, mode = *Par, Ind*). *x.a* is the offset from the stack pointer.

Before an operator can be applied to operands, these must first be transferred (loaded) into registers. This is because the RISC performs operations only on registers. The loading is achieved by procedure *load* (and *loadAdr*) in ORG. The resulting mode is *Reg*. In allocating registers, a strict stack principle is used, starting with R0, up to R11. This is certainly not an optimal strategy and provides ample room for improvement (usually called optimization). The compiler variable RH indicates the next free register (top of register stack).

Base address SB is, as the name suggests, static. But this holds only within a module. It implies that on every transfer to a procedure in another module, the static base must be adjusted. The simplest way is to load SB before every external call, and to restore it to its old value after return from the procedure. We chose a different strategy: loading on demand (see below: global variables).

If a variable is indexed, has a field selector, is dereferenced, or has a type guard, this is detected in the parser by procedure *selector*. It calls generators *Index, Field, DeRef,* or *TypeTest* accordingly (see Section 12.3.2. and patterns 1 - 4 in Section 12.2). These procedures cause item modes to change as follows:

mode transition of x	instructions emitted	construct
1. Index(x, y) (y is loaded into y.r)		
Var --> RegI	ADD y.r, SP, y.r	array variable
Par --> RegI	LDR RH, SP, x.a ADD y.r, RH, y.r	array parameter
RegI --> RegI	ADD x.r, x.r, y.r	indexed array
2. Field(x, y) (y.mode = Fld, y.a = field offset)		
Var --> Var	none	field designator, add offset to x.a
RegI --> RegI	none	add field offset to x.a
Par --> Par	none	add field offset to x.b
3. DeRef(x)		
Var --> RegI	LDR RH, SP, x.a	dereferenced x^
Par --> RegI	LDR RH, SP, x.a LDR RH, RH, x.b	dereferenced parameter x^
RegI --> RegI	LDR x.r, x.r, x.a	

A fairly large number of procedures then deal with individual operators. Specifically, they are *Not, And1, And2, Or1, Or2* for Boolean operators, *Neg, AddOp, MulOp, DivOp* for operations on integers, *RealOp* for operations on real numbers, and *Singleton, Set, In, and SetOp* for operations on Sets. And finally, following the same pattern, are the procedures for relations (comparisons) *IntRelation, SetRelation, RealRelation, StringRelation.* (see Appendix for listing of ORG). We note in particular that if all operands are constants, their evaluation is performed by the compiler and not delegated to run-time. This is an important efficiency factor.

12.7.2. Relations

RISC does not feature any compare instruction. Instead, subtraction is used, because an implicit comparison wth 0 is performed along with any arithmetic (or load) instruction. Instead of *x* < *y* we use *x-y* < 0. This is possible, because in addition to the computed difference deposited in a register, also the result of the comparison is deposited in the condition flags *N* (difference negative) and *Z* (difference zero). Relations therefore yield a result item *x* with mode *Cond*. *x.r (= relmap[sym])* identifies the relation. Branch instructions (jumps) are executed conditionally depending on these flags. The value *x.r* is then used when generating branch instructions. For example, the relation *x* < *y* is translated simply into

```
LDR  R0, SP, x
LDR  R1, SP, y
CMP  R0, R0, R1
```

and the resulting item mode is *x.mode* = *Cond, x.r := "less"*. (The mnemonic CMP is synonymous with SUB). More about relations and Boolean expressions will be explained in Section 12.7.6.

12.7.3. Set operations

The type SET represents sets of small integers in the range from 0 to 31. Bit *i* being 1 signals that *i* is an element of the set. This is a convenient representation, because the logical instructions directly mirror the set operations: AND implements set intersection, OR set union, and XOR the symmetric set difference. This representation also allows a simple and efficient implementation of membership tests. The instructions for the expression *n IN s* is generated by procedure *In*. Assuming the value *n* in register R0, and the set *s* in R1, we obtain

```
ADD  R0, R0, 1
ROR  R1, R1, R0     rotate s by i+1 position, the relevant bit moving to the sign bit
```

The resulting item mode is Cond with x.r = "minus".

Of some interest are the procedures for generating sets, i.e. for processing {*m*}, {*m .. n*}, and {*m, n*}, where *m, n* are integer expressions.

We start with {*m*}. It is generated by procedure *Singleton* using a shift instruction. Assuming *m* in R0, the resulting code is

```
MOV  R1, 0, 1
LSL  R0, R1, R0   shift 1 by m bit positions to the left
```

Somewhat more sophisticated is the generation of {*m .. n*} by procedure *Set*. Assuming *m* in R0, and *n* is R1, the resulting code is

```
MOV  R2, 0, -2
LSL  R1, R2, R1   shift -2 by n bit positions to the left
MOV  R2, 0, -1
LSL  R0, R2, R0   shift -1 by m bit positions to the left
XOR  R0, R0, R1
```

The set {*m, n*} is generated as the union of {*m*} and {*n*}. If any of the element values is a constant, several possibilities of code improvement are possible. For details, the reader is referred to the source code of ORG.

12.7.4. Assignments

Statements have an effect, but no result like expressions. Statements are executed, not evaluated. Assignments alter the value of variables through store instructions. The computation of the address of the affected variable follows the same scheme as for loading. The value to be assigned must be in a register.

Assignments of arrays (and records) are an exceptional case in so far as they are performed not by a single store instruction, but by a repetition. Consider y := x, where x, and y are both arrays of *n* integers. Assuming that the address of *y* is in register R0, that of *x* in R1, and the value *n* in R2. Then the resulting code is

```
L       LDR  R3, R1, 0   source
        ADD  R1, R1, 4
        STR  R3, R0, 0   destination
        ADD  R0, R0, 4
        SUB  R2, R2, 1   counter
        BNE  L
```

12.7.5. Conditional and repetitive statements

These statements are implemented using branch instructions (jumps) as shown in Section 12.2, Patterns 5 - 7. In all repetitive statements, backward jumps occur. Here, at the point of return the value of the global variable ORG.pc is saved in a local (!) variable of the involved parsing procedure. It is retrieved when the backward jump is emitted. We note that branch instructions use a displacement rather than an absolute destination address. It is the difference between the branch instruction and the destination of the jump.

A difficulty, however, arises in the case of forward jumps, a difficulty inherent in all single-pass compilers: When the branch is issued, its destination is still unknown. It follows that the branch displacement must be later inserted when it becomes known, when the destination is reached. This is called a *fixup*. Here the method of fixup lists is used. The place of the instruction with still unknown destination is held in a variable *L* local to the respective parsing procedure. If several branches have the same destination, L is the heading of a list of the instructions to be fixed up, with its links placed in the instructions themselves in the place of the eventual jump displacement. This shown for the if statement by an excerpt of *ORP.StatSequence* with local variable L0:

```
ELSIF sym = ORS.if THEN
  ORS.Get(sym); expression(x); ORG.CFJump(x);
  StatSequence; L0 := 0;
  WHILE sym = ORS.elsif DO
    ORS.Get(sym); ORG.FJump(L0); ORG.Fixup(x); expression(x);
    ORG.CFJump(x); Check(ORS.then, "no THEN"); StatSequence
  END ;
  IF sym = ORS.else THEN ORS.Get(sym); ORG.FJump(L0); ORG.Fixup(x); StatSequence
  ELSE ORG.Fixup(x)
  END ;
  ORG.FixLink(L0);
```

where in module ORG:

```
PROCEDURE CFJump(VAR x: Item);  (*conditional forward jump*)
BEGIN
  IF x.mode # Cond THEN loadCond(x) END ;
  Put3(BC, negated(x.r), x.a); FixLink(x.b); x.a := pc-1
END CFJump;

PROCEDURE FJump(VAR L: LONGINT);  (*unconditional forward jump*)
BEGIN Put3(BC, 7, L); L := pc-1
END FJump;

PROCEDURE fix(at, with: LONGINT);
BEGIN code[at] := code[at] DIV C24 * C24 + (with MOD C24)
END fix;
```

```
PROCEDURE FixLink(L: LONGINT);
   VAR L1: LONGINT;
BEGIN invalSB;
   WHILE L # 0 DO L1 := code[L] MOD 40000H; fix(L, pc-L-1); L := L1 END
END FixLink;

PROCEDURE Fixup(VAR x: Item);
BEGIN FixLink(x.a)
END Fixup;
```

In while-, repeat-, and for statements essentially the same technique is used with the support of the identical procedures in ORG.

12.7.6. Boolean expressions

In the case of arithmetic expressions, our compilation scheme results in a conversion from infix to postfix notation (x+y => xy+). This is not applicable for Boolean expressions, because the operators & and OR are defined as follows:

```
x & y  -->  if x then y else FALSE
x OR y  -- >  if x then TRUE else y
```

This entails that depending on the value of *x*, *y* must not be evaluated. As a consequence, jumps may have to be taken across the code for *y*. Therefore, the same technique of conditional evaluation must be used as for conditional statements. In the case of an expression *x & y (x OR y),* procedure ORG.And1 resp. ORG.Or1 must be called just after parsing *x* (see ORP.term resp. ORP.SimpleExpression). Only after parsing also *y* can the generators ORG.And2 resp. ORG(Or2) be called, providing the necessary fixups of forward jumps.

```
PROCEDURE And1(VAR x: Item);   (* x := x & *)
BEGIN
   IF x.mode # Cond THEN loadCond(x) END ;
   Put3(BC, negated(x.r), x.a); x.a := pc-1; FixLink(x.b); x.b := 0
END And1;

PROCEDURE And2(VAR x, y: Item);
BEGIN
   IF y.mode # Cond THEN loadCond(y) END ;
   x.a := merged(y.a, x.a); x.b := y.b; x.r := y.r
END And2;
```

A negative consequence of this scheme having condition flags in the processor is that when an item with mode *Cond* has to be transferred into mode *Reg*, as in a Boolean assignment, an unpleasantly complex instruction sequence must be generated. Fortunately, this case occurs quite rarely.

12.7.7. Procedures

Before embarking on an explanation of procedure calls, entries and exits, we need to know how recursion is handled and how storage for local variables is allocated. Procedure calls cause a sequence of frames to be allocated in a stack fashion. These frames are the storage space for local variables. Each frame is headed by a single word containing the return address of the call. This address is deposited in R15 by the call instructions (BL, branch and link). The compiler "knows" the size of the frame to be allocated, and thus merely decrements the stack pointer SP (R14) by this amount. Upon return, SP is incremented by the same amount, and PC is restored by a branch instruction. In the following example, a procedure P is called, calling itself Q, and Q calling P again (recursion). The stack then contains 3 frames (see Figure 12.7).

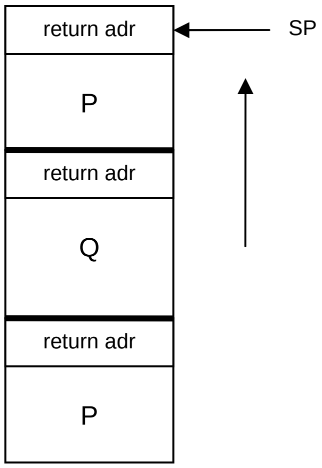


Figure 12.7 Stack frames

Scheme and layout determine the code sequences for call, entry and exit of procedures. Here is an example of a procedure *P* with 2 parameters:

```
Call:      LDR  R0, param0
           LDR  R1, param1
           BL   P

Prolog:    SUB  SP, SP, size      decrement SP
           STR  LNK, SP, 0        push return adr
           STR  R0, SP, 4         push parameter 0
           STR  R1, SP, 8         push parameter1 ....

Epilog:    LDR  LNK, SP, 0        pop return adr
           ADD  SP, SP, size      increment SP
           BR  LNK
```

When the call instruction is executed, parameters reside in registers, starting with R0. For function procedures, the result is passed in register R0. This scheme is very efficient; storing the parameters occurs only in a single place, namely at procedure entry, and not before each call. However, it has severe consequences for the entire register allocation strategy. Throughout the compiler, registers *must* be allocated in strict stack fashion. Furthermore, parameter allocation *must* start with R0. This is a distinct drawback for function calls. If registers are occupied by other values loaded prior to the call, they must be cleared, i.e. the parameters must be saved and reloaded after return. This is rather cumbersome (see procedures *ORG.SaveRegisters* and *ORG.RestoreRegisters*).

F(x)	no register saving
x + F(x)	
F(F(x))	
(x+1) + F(x)	register saving necessary

12.7.8. Type extension

Static typing is an important principle in programming languages. It implies that every constant, variable or function is of a certain data type, and that this type can be derived by reading the program text without executing it. It is the key principle to introduce important redundancy in languages in such a form that a compiler can detect inconsistencies. It is therefore the key element for reducing the number of errors in programs.

However, it also acts as a restriction. It is, for example, impossible to construct data structures (arrays, trees) with different types of elements. In order to relax the rule of strictly static typing, the notion of *type extension* was introduced in Oberon. It makes it possible to construct inhomogeneous data structures without abandoning type safety. The price is that the checking of type consistency must in certain instances be deferred to run-time. Such checks are called *type tests*. The challenge is to defer to run-time as few checks as possible and as many as needed.

The solution in Oberon is to introduce families of types, and compatibility among their members. Their members are thus related, and a family forms a hierarchy. The principle idea is the following: Any record type T0 can be extended into a new type T1 by additional record fields (attributes). T1 is then called an *extension* of T0, which in turn is said to be T1's *base type*. T1 is then type compatible with T0, but not vice-versa. This property ensures that in many cases static type checking is still possible. Furthermore, it turns out that run-time tests can be made very efficient, thus minimizing the overhead for maintaining type safety.

For example, given the declarations

```
TYPE R0 = RECORD u, v: INTEGER END ;
     R1 = RECORD (R0) w: INTEGER END
```

we say that R1 is an *extension* of R0. R0 has the fields u and v, R1 has u, v, and w. The concept becomes useful in combination with pointers. Let

```
TYPE P0 = POINTER TO R0;
     P1 = POINTER TO R1;
VAR p0: P0;  p1: P1;
```

Now it is possible to assign p1 to p0 (because a P1 is always also a P0), but not p0 to p1, because a P0 need not be a P1. This has the simple consequence that a variable of type P0 may well point to an extension of R0. Therefore, data structures can be declared with a base type, say P0, as common element type, but in fact they can individually differ, they can be any extension of the base type.

Obviously, it must be possible to determine the actual, current type of an element even if the base type is statically fixed. This is possible through a *type test*, syntactically a Boolean factor:

```
p0 IS P1                    (short for p0^ IS R1)
```

Furthermore, we introduce the *type guard*. In the present example, the designator p0.w is illegal, because there is no field *w* in a record of type R0, even if the current value of p0^ is a R1. As this case occurs frequently, we introduce the short notation *p0(P1).w*, implying a test *p0 IS P1* and an abort if the test is not met.

It is important to mention that this technique also applies to formal variable parameters of record type, as they also represent a pointer to the actual parameter. Its type may be any extension of the type specified for the formal parameter in the procedure heading.

How are type test and type guard efficiently implemented? Our first observation is that they must consist of a single comparison only, similar to index checks. This in turn implies that types must be identified by a single word. The solution lies in using the unique address of the *type descriptor* of the (record) type. Which data must this descriptor hold? Essentially, type descriptors (TD) must identify the base types of a given type. Consider the following hierarchy:

```
TYPE T =  RECORD … END ;
     T0 =  RECORD (T) … END ;        extension level 1
     T1 =  RECORD (T) … END ;        extension level 1
     T00 = RECORD (T0) … END ;       extension level 2
     T01 = RECORD (T0) … END ;       extension level 2
     T10 = RECORD (T1) … END ;       extension level 2
     T11 = RECORD (T1) … END ;       extension level 2
```

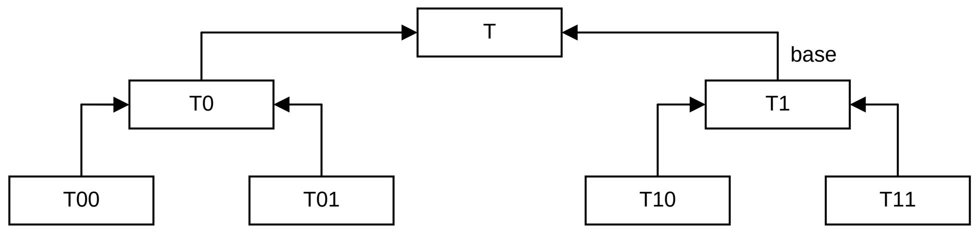


Figure 12.8. A type hierarchy

In the symbol table, the field *base* refers to the ancestor of a given record type. Thus base of the type representing T11 points to T1, etc. Run-time checks, however, must be fast, and hence cannot proceed through chains of pointers. Instead, each TD contains an array with references to the ancestor TDs (including itself). For the example above, the TDs are as follows:

```
TD(T)  =   [T]
TD(T0) =   [T, T0]
TD(T1) =   [T, T1]
TD(T00) =  [T, T0, T00]
TD(T01) =  [T, T0, T01]
TD(T10) =  [T, T1, T10]
TD(T11) =  [T, T1, T11]
```

Evidently, the first element can be omitted, as it always refers to the common base of the type hierarchy. The last element always points to the TD's owner. TDs are allocated in the data area, the area for variables.

References to TDs are called *type tags*. They are required in two cases. The first is for records referenced by pointers. Such dynamically allocated records carry an additional, hidden field holding their type tag. (A second additional word is reserved for use by the garbage collector. The offset of the tag field is therefore -8). The second case is that of record-typed VAR-parameters. In this case the type tag is explicitly passed along with the address of the actual parameter. Such parameters therefore require two words/registers.

A type test then consists of a test for equality of two type tags. In *p IS T* the first tag is that of the n'th entry of the TD of *p^*, where n is the extension level of *T*. The second tag is that of type *T*. This is shown in Pattern13 in Section 12.2 (see also Fig. 12.4). The test then is as follows:

```
p^.tag^[n] = adr(T),  where n is the extension level of T
```

When declaring a record type, it is not known how many extensions, nor how many levels will be built on this type. Therefore TD's should actually be infinite arrays. We decided to restrict them to 3 levels only. The first entry, which is never used for checking, is replaced by the size of the record.

12.7.9. Import and export, global variables

Addresses of imported objects are not available to the compiler. Their computation must be left to the module loader (see Chapter 6). Similar to handling addresses of forward jumps, the compiler puts the necessary information in place of the actual address into the instruction itself. In the case of procedure calls, this is quite feasible, because the BL instruction features an offset field of 24 bits. The information consists of the module number and the export number of the imported object. In addition, there is a link to the previous instruction referring to an imported procedure. The origin of the list of procedure call fixups is rooted in the compiler variable *fixorgP*, and of the 24 bits in each BL instruction 4 bits are used for the module number, 8 bits for the object's export number, and 12 for the link. The loader need only scan this list to fix up the addresses (jump offsets).

Matters are more complex in the case of data. Object records in the symbol table have a field *lev*. It indicates the nesting level of variables local to procedures. It is also used for the module number in the case of variables of imported modules. Note that when importing, objects designating modules are inserted in the symbol table, and the list of their own objects are attached in the field *dsc*. In this latter case, the module numbers have an inverted sign (are negative). Such imported objects are static, i.e. have a fixed address. In principle their absolute address could be computed (fixed) by the module loader. However, this is not practicable, because RISC instructions have an address offset of 16 bits only. It is therefore necessary in the general case to use a base address in conjunction with the offset. We use a single register for holding the *static base* (SB, R13). This register need be reloaded for every access to an imported variable. However, the compiler keeps track of external accesses; if a variable is to be accessed from the same module as the previous case, then reloading is avoided (see procedure *GetSB* and global compiler variable *curSB*).

This base address is fetched from a table global to the entire system. This module table contains one entry for every module loaded, namely the address of the module's data section. The address of the table is permanently in register MT (= R12). An access to an imported variable therefore always requires two instructions:

```
LDR  SB, MT, modno*4     base address of data section
LDR  R0, SB, offset      offset computed by the loader from object's export number
```

Considering the fact that references to external variables are (or should be) rare, this circumstance is of no great concern. (Note also that such accesses are read-only). More severe is the fact that we also treat global variables contained in the same module by the same technique. Their level number is 0. One might use a specific base register for the base of the current module. Its content would then have to be reloaded upon every procedure call and after every return. This is common technique, but we have here chosen to reload only when necessary, i.e. only when an access is at hand. This strategy rewards the programmer who sensibly uses global variables rarely.

12.7.10. Traps

This compiler provides an extensive system of safeguard by providing run-time checks (aborts) in several cases:

trap number	trap cause
1	array index out of range
2	type guard failure
3	array or string copy overflow
4	access via NIL pointer
5	illegal procedure call
6	integer division by zero
7	assertion violated

These checks are implemented very efficiently in order not to downgrade a program's performance. Involved is typically a single compare instruction, plus a conditional branch (BLR MT). It is assumed that entry 0 of the module table contain not a base address (module numbers start with 1), but a branch instruction to an appropriate trap routine. The trap number is encoded in bits 4:7 of the branch instruction.

The predefined procedure *Assert* generates a conditional trap with trap number 7. For example, the statement *Assert(m = n)* generates

```
LDR     R0, m
LDR     R1, n
CMP     R0, R0, R1
BLR     1, 7CH          branch and link if unequal through R12, trap number 7
```

Procedure *New*, representing the operator *NEW*, has been implemented with the aid of the trap mechanism. (This is in order to omit in ORG any reference to module *Kernel*, which contains the allocation procedure *New*). The generated code for the statement NEW(p) is

```
ADD     R0, SP, p       address of p
ADD     R1,SB,  tag     type tag
BLR     7, 0CH          branch and link unconditionally through R12 (MT), trap number 0
```

13 A graphics editor

13.1. History and goal

The origin of graphics systems as they are in use at this time was intimately tied to the advent of the high-resolution bit-mapped display and of the mouse as pointing device. The author's first contact with such equipment dates back to 1976. The Alto computer at the Xerox Palo Alto Research Center is justly termed the first workstation featuring those characteristics. The designer of its first graphics package was Ch. Thacker who perceived the usefulness of the high-resolution screen for drawing and processing schematics of electronic circuits. This system was cleverly tailored to the needs encountered in this activity, and it was remarkable in its compactness and effectiveness due to the lack of unnecessary facilities. Indeed, its acronym was SIL, for Simple ILlustrator.

After careful study of the used techniques, the author designed a variant, programmed in Modula-2 (instead of BCPL) for the PDP-11 Computer, thereby ordering and exhibiting the involved data structures more explicitly. In intervals of about two years, that system was revised and grew gradually into the present Draw system. The general goal remained a simple line drawing system: emphasis was placed on a clear structure and increase of flexibility through generalization of existing rather than indiscriminate addition of new features.

In the history of this evolution, three major transitions can be observed. The first was the move from a single "window", the screen, to multiple windows including windows showing different excerpts of the same graphic. This step was performed on the Lilith computer which resembled the Alto in many ways. The second major transition was the application of the object-oriented style of programming, which allowed the addition of new element types to the basic system, making it extensible. The third step concerned the proper integration of the Draw system with Oberon's text system. The last two steps were performed using Oberon and the Ceres computer.

We refrain from exhibiting this evolution and merely present the outcome, although the history might be an interesting reflection of the evolution of programming techniques in general, containing many useful lessons. We stress the fact, however, that the present system rests on a long history of development, during which many features and techniques were introduced and later discarded or revised. The size of the system's description is a poor measure of the effort that went into its construction; deletion of program text sometimes marks bigger progress than addition.

The goal of the original SIL program was to support the design of electronic circuit diagrams. Primarily, SIL was a line drawing system. This implies that the drawings remain uninterpreted. However, in a properly integrated system, the addition of modules containing operators that interpret the drawings is a reasonably straight-forward proposition. In fact, the Oberon system is ideally suited for such steps, particularly due to its command facility.

At first, we shall ignore features specially tailored to circuit design. The primary one is a macro facility to be discussed in a later chapter.

The basic system consists of the modules *Draw, GraphicFrames,* and *Graphics*. These modules contain the facilities to generate and handle horizontal and vertical lines, text captions, and macros. Additional modules serve to introduce other elements, such as rectangles and circles, and the system is extensible, i.e. further modules may be introduced to handle further types of elements.

13.2. A brief guide to Oberon's line drawing system

In order to provide the necessary background for the subsequent description of the Draw system's implementation, a brief overview is provided in the style of a user's guide. It summarizes the facilities offered by the system and gives an impression of its versatility.

The system called *Draw* serves to prepare line drawings. They contain lines, text captions, and other items, and are displayed in graphic viewers (more precisely: in menu viewers' graphic frames). A graphic viewer shows an excerpt of the drawing plane, and several viewers may show different parts of a drawing. The most frequently used commands are built-in as mouse clicks and combinations of clicks. Additional commands are selectable from texts, either in viewer menus (title bars) or in the text called *Draw.Tool*. Fig. 13.1. shows the display with two graphic viewers at the left and the draw tool text at the right. The mouse buttons have the following principal functions whenever the cursor lies in a graphic frame:

left:	draw / set caret
middle:	move / copy
right:	select

A mouse command is identified (1) by the key k0 pressed initially, (2) by the initial position P0 of the cursor, (3) by the set of pressed keys k1 until the last one is released, and (4) the cursor position P1 at the time of release.

13.2.1. Basic commands

The command *Draw.Open* opens a new viewer and displays the graph with the name given as parameter. We suggest that file names use the extension *Graph*.

Drawing a line. In order to draw a horizontal or vertical line from P0 to P1, the left key is pressed with the cursor at P0 and, while the key is held, the mouse and cursor is moved to P1. Then the key is released. If P0 and P1 differ in both their x and y coordinates, the end point is adjusted so that the line is either horizontal or vertical.

Writing a caption. First the cursor is positioned where the caption is to appear. Then the left key is clicked, causing a crosshair to appear. It is called the *caret*. Then the text is typed. Only single lines of texts are accepted. The DEL key may be used to retract characters (backspace).

Selecting. Most commands require the specification of operands, and many implicitly assume the previously selected elements - the *selection* - to be their operands. A single element is selected by pointing at it with the cursor and then clicking the right mouse button. This also causes previously selected elements to be deselected. If the left key is also clicked, their selection is retained. This action is called an *interclick*. To select several elements at once, the cursor is moved from *P0* to *P1* while the right key is held. Then all elements lying within the rectangle with diagonally opposite corners at *P0* and *P1* are selected. Selected lines are displayed as dotted lines, selected captions (and macros) by inverse video mode. A macro is selected by pointing at its lower left corner. The corner is called *sensitive area*.

Moving. To move (displace) a set of elements, the elements are first selected and then the cursor is moved from P0 to P1 while the middle key is held. The vector from *P0* to *P1* specifies the movement and is called the *displacement vector*. *P0* and *P1* may lie in different viewers displaying the same graph. Small displacements may be achieved by using the keyboard's cursor keys.

Copying. Similarly, the selected elements may be copied (duplicated). In addition to pressing the middle key while indicating the displacement vector, the left key is interclicked. The copy command may also be used to copy elements from one graph into another graph by moving the cursor from one viewer into another viewer displaying the destination graph. A text caption may be copied from a text frame into a graphic frame and vice-versa. There exist two ways to accomplish this: 1. First the caret is placed at the destination position, then the text is selected and the middle key is interclicked. 2. First the text is selected, then the caret is placed at the destination position and the middle key is interclicked.

Shifting the plane. You may shift the entire drawing plane behind the viewer by specifying a displacement vector pressing the middle button (like in a move command) and interclicking the right button.

The following table shows a summary of the mouse actions:

left	draw line
left (no motion)	set caret

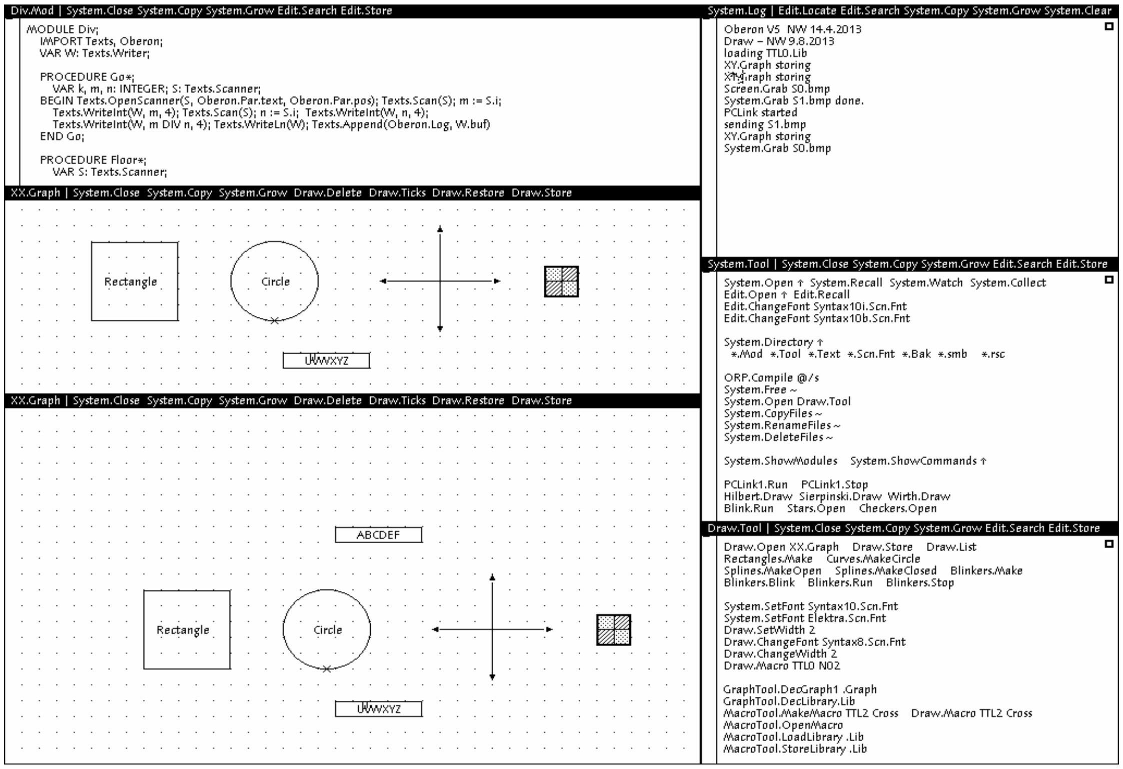
left + middle	copy selected caption to caret
left + right	set secondary caret
middle	move selection
middle + left	copy selection
middle + right	shift drawing plane
right	select area
right (no motion)	select object
right + middle	copy caption to caret
right + left	select without deselection

13.2.2. Menu commands

The following commands are displayed in the menu (title bar) of every graphic viewer. They are activated by being pointed at and by clicking the middle button.

Draw.Delete	The selected elements are deleted.
Draw.Store	The drawing is written as file with the name shown in the title bar. The original file is renamed by appending ".Bak".
Draw.Restore	The entire frame is redrawn
Draw.Ticks	The frame displays a pattern of dots (ticks) to facilitate positioning.

The two viewers in Fig. 13.1. display different parts of the same graphic. The second view was obtained from the generic *System.Copy* command and a subsequent shift of the drawing plane.


Div.Mod | System.Close System.Copy System.Grow Edit.Search Edit.Store
XX.Graph | System.Close System.Copy System.Grow Draw.Delete Draw.Ticks Draw.Restore Draw.Store
Rectangle
Circle
UVWXYZ
XX.Graph | System.Close System.Copy System.Grow Draw.Delete Draw.Ticks Draw.Restore Draw.Store
ABCDEF
Rectangle
Circle
UVWXYZ
System.Log | Edit.Locate Edit.Search System.Copy System.Grow System.Clear
System.Tool | System.Close System.Copy System.Grow Edit.Search Edit.Store
Draw.Tool | System.Close System.Copy System.Grow Edit.Search Edit.Store

```
MODULE Div;
  IMPORT Texts, Oberon;
  VAR W: Texts.Writer;

  PROCEDURE Go*;
    VAR k, m, n: INTEGER; S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S); m := S.i;
    Texts.WriteInt(W, m, 4); Texts.Scan(S); n := S.i;  Texts.WriteInt(W, n, 4);
    Texts.WriteInt(W, m DIV n, 4); Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END Go;

  PROCEDURE Floor*;
    VAR S: Texts.Scanner;
```

```
Oberon V5  NW 14.4.2013
Draw – NW 9.8.2013
loading TTL0.Lib
XY.Graph storing
XY.Graph storing
Screen.Grab S0.bmp
System.Grab S1.bmp done.
PCLink started
sending S1.bmp
XY.Graph storing
System.Grab S0.bmp
```

```
System.Open ↑  System.Recall  System.Watch  System.Collect
Edit.Open ↑  Edit.Recall
Edit.ChangeFont Syntax10i.Scn.Fnt
Edit.ChangeFont Syntax10b.Scn.Fnt

System.Directory ↑
 *.Mod  *.Tool  *.Text  *.Scn.Fnt  *.Bak  *.smb   *.rsc

ORP.Compile @/s
System.Free ~
System.Open Draw.Tool
System.CopyFiles ~
System.RenameFiles ~
System.DeleteFiles ~

System.ShowModules   System.ShowCommands ↑

PCLink1.Run   PCLink1.Stop
Hilbert.Draw  Sierpinski.Draw  Wirth.Draw
Blink.Run   Stars.Open   Checkers.Open
```

```
Draw.Open XX.Graph   Draw.Store   Draw.List
Rectangles.Make   Curves.MakeCircle
Splines.MakeOpen   Splines.MakeClosed   Blinkers.Make
Blinkers.Blink   Blinkers.Run   Blinkers.Stop

System.SetFont Syntax10.Scn.Fnt
System.SetFont Elektra.Scn.Fnt
Draw.SetWidth 2
Draw.ChangeFont Syntax8.Scn.Fnt
Draw.ChangeWidth 2
Draw.Macro TTL0 N02

GraphTool.DecGraph1 .Graph
GraphTool.DecLibrary.Lib
MacroTool.MakeMacro TTL2 Cross   Draw.Macro TTL2 Cross
MacroTool.OpenMacro
MacroTool.LoadLibrary .Lib
MacroTool.StoreLibrary .Lib
```

Figure 13.1 Display with graphics duplicated viewers

13.2.3. Further commands

The following commands are listed in the text *Draw.Tool*, but may appear in any text.

Draw.Store *name*	The drawing in the marked viewer is stored as a file with the specified name.

The subsequent commands change attributes of drawing elements, such as line width, text font, and color, and they apply to the most recent selection.

Draw.SetWidth *w*	default = 1, 0 < *w* < 7.
Draw.ChangeFont *fontname*	
Draw.ChangeColor *c*	
Draw.ChangeWidth *w*	(0 < *w* < 7)

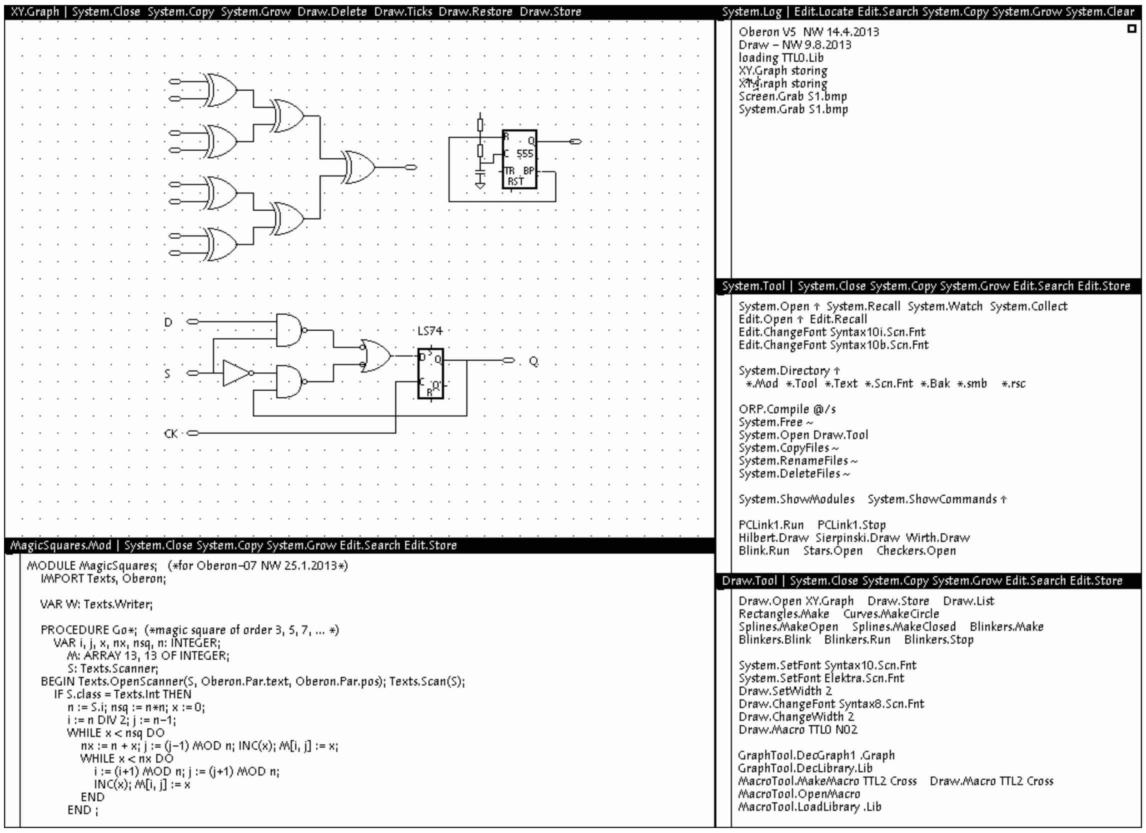
The *ChangeColor* command either take a color number in the range 1 .. 15 or a string as parameter. It serves to copy the color from the selected character.

13.2.4. Macros

A macro is a (small) drawing that can be identified as a whole and be used as an element within a (larger) drawing. Macros are typically stored in collections called *libraries*, from where they can be selected and copied individually.

Draw.Macro *lib mac*	The macro *mac* is selected from the library named *lib* and inserted in the drawing at the caret's position.

An example for the use of macros is drawing electronic circuit diagrams. The basic library file containing frequently used TTL components is called *TTL0.Lib*, and a drawing showing its elements is called *TTL0.Graph* (see Figure 13.2).


XY.Graph | System.Close System.Copy System.Grow Draw.Delete Draw.Ticks Draw.Restore Draw.Store
R
Q
C
555
TR
BP
RST
D
S
CK
LS74
Q
S
Q
D
C
R
System.Log | Edit.Locate Edit.Search System.Copy System.Grow System.Clear
System.Tool | System.Close System.Copy System.Grow Edit.Search Edit.Store
Draw.Tool | System.Close System.Copy System.Grow Edit.Search Edit.Store
MagicSquares.Mod | System.Close System.Copy System.Grow Edit.Search Edit.Store

```
Oberon V5  NW 14.4.2013
Draw – NW 9.8.2013
loading TTL0.Lib
XY.Graph storing
XY.Graph storing
Screen.Grab S1.bmp
System.Grab S1.bmp
```

```
System.Open ↑  System.Recall  System.Watch  System.Collect
Edit.Open ↑  Edit.Recall
Edit.ChangeFont Syntax10i.Scn.Fnt
Edit.ChangeFont Syntax10b.Scn.Fnt

System.Directory ↑
 *.Mod  *.Tool  *.Text  *.Scn.Fnt  *.Bak  *.smb   *.rsc

ORP.Compile @/s
System.Free ~
System.Open Draw.Tool
System.CopyFiles ~
System.RenameFiles ~
System.DeleteFiles ~

System.ShowModules   System.ShowCommands ↑

PCLink1.Run   PCLink1.Stop
Hilbert.Draw  Sierpinski.Draw  Wirth.Draw
Blink.Run   Stars.Open   Checkers.Open
```

```
Draw.Open XY.Graph   Draw.Store   Draw.List
Rectangles.Make   Curves.MakeCircle
Splines.MakeOpen   Splines.MakeClosed   Blinkers.Make
Blinkers.Blink   Blinkers.Run   Blinkers.Stop

System.SetFont Syntax10.Scn.Fnt
System.SetFont Elektra.Scn.Fnt
Draw.SetWidth 2
Draw.ChangeFont Syntax8.Scn.Fnt
Draw.ChangeWidth 2
Draw.Macro TTL0 N02

GraphTool.DecGraph1 .Graph
GraphTool.DecLibrary.Lib
MacroTool.MakeMacro TTL2 Cross   Draw.Macro TTL2 Cross
MacroTool.OpenMacro
MacroTool.LoadLibrary .Lib
```

```
MODULE MagicSquares;   (*for Oberon-07 NW 25.1.2013*)
  IMPORT Texts, Oberon;

  VAR W: Texts.Writer;

  PROCEDURE Go*;  (*magic square of order 3, 5, 7, ... *)
    VAR i, j, x, nx, nsq, n: INTEGER;
      M: ARRAY 13, 13 OF INTEGER;
      S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
    IF S.class = Texts.Int THEN
      n := S.i; nsq := n*n; x := 0;
      i := n DIV 2; j := n-1;
      WHILE x < nsq DO
        nx := n + x; j := (j-1) MOD n; INC(x); M[i, j] := x;
        WHILE x < nx DO
          i := (i+1) MOD n; j := (j+1) MOD n;
          INC(x); M[i, j] := x
        END
      END ;
```

Figure 13.2 Viewer with circuit macros of TTL0 library

13.2.5. Rectangles

Rectangles can be created as individual elements. They are frequently used for framing sets of elements. Rectangles consist of four lines which are selectable as a unit. The attribute commands *Draw.SetWidth, System.SetColor, Draw.ChangeWidth,* and *Draw.ChangeColor* also apply to rectangles. Rectangles are selected by pointing at their lower left corner and are created by the following steps:

1. The caret is placed where the lower left corner of the new rectangle is to lie.
2. A secondary caret is placed where the opposite corner is to lie (ML + MR).
3. The command *Rectangles.Make* is activated.

13.2.6. Oblique lines, circles, and ellipses

Further graphic elements are (oblique) lines, circles, and ellipses. The sensitive area of circles and ellipses is at their lowest point. They are created by the following steps:

Lines:	1. The caret is placed where the starting point is to lie. 2. A secondary caret is placed at the position of the end. 3. The command *Curves.MakeLine* is activated.
Circles:	1. The caret is placed where the circle's center is to lie. 2. A secondary caret is placed, its distance from the center specifying the radius. 3.The command *Curves.MakeCircle* is activated.
Ellipses:	1. The caret is placed where the center is to lie. 2. A second caret is placed. Its horizontal distance from the first caret specifies one axis. 3. A third caret is placed. Its vertical distance from the first caret specifies the other axis. 4. The command *Curves.MakeEllipse* is activated.

13.2.7. Spline curves

Spline curves are created by the following steps:

1. The caret is placed where the starting point is to lie.
2. Secondary carets are placed at the spline's fixed points (at most 20).
3. The command *Splines.MakeOpen* or *Splines.MakeClosed* is activated.

13.2.8. Constructing new macros

A new macro is constructed and inserted in the library lib under the name mac as follows:

1. All elements which belong to the new macro are selected.
2. The caret is placed at the lower left corner of the area to be spanned by the macro.
3. A secondary caret is placed at the upper right corner of the area to be spanned.
4. The command *MacroTool.MakeMacro lib mac* is activated.

An existing macro can be decomposed (opened) into its parts as follows:

1. The macro is selected.
2. The caret is placed at the position where the decomposition is to appear.
3. The command *MacroTool.OpenMacro* is activated.

The command *MacroTool.StoreLibrary lib file* stores the library *lib* on the specified file. Only the macros presently loaded are considered as belonging to the library. If one wishes to add some macros to an existing library file, all of its elements must first be loaded. This is best done by opening a graph containing all macros of the desired library file.

13.3. The core and its structure

Like a text, a graphic consists of elements, subsequently to be called *objects*. Unlike a text, which is a sequence of elements, a graphic is an unordered set of objects. In a text, the position of an element need not be explicitly indicated (stored); it is recomputed from the position of its predecessor each time it is needed, for example for displaying or selecting an element. In a graphic, each object must carry its position explicitly, as it is independent of any other object in the set. This is an essential difference, requiring a different treatment and much more storage space for an equal number of objects.

Although this is an important consideration in the choice of a representation of a data structure, the primary determinants are the kind of objects to be included, and the set of operations to be applied to them. Here SIL set a sensible starting point. To begin with, there exist only two kinds of objects, namely straight, horizontal and vertical lines, and short texts for labelling lines, called *captions*. It is surprising how many useful task can be fulfilled with only these two types of objects.

The typical operations to be performed on objects are creating, drawing, moving, copying, and erasing. Those performed on a graphic are inserting, searching, and deleting an object. For the operations on objects, data indicating an object's position (and possibly color), its length and width in the case of lines, and the character string in the case of captions suffice. For the operations on the graphic, some data structure representing the set of objects must be chosen. Without question, a dynamic structure is most appropriate, and it requires the addition of some linking fields to the record representing an object. Without further deliberation, and with the idea that graphics to be handled with this system contain hundreds rather than tens of thousands of objects, we chose the simplest solution, the linear list. A proper modularization in connection with information hiding will make it possible to alter this choice without affecting client modules.

Although in general the nature of a user interface should not influence the representation chosen for the abstract data structure, we need to take note of the manner in which parameters of certain operations are denoted. It is, for example, customary in interactive graphics systems to select the objects to which an operation is to apply *before* invoking that operation. Their *selection* is reflected in their visual appearance in some way, and gives the user an opportunity to verify the selection (and to change it, if necessary) before applying the operation (such as deletion). For an object to be selectable means that it must record a state (selected/unselected). We note that it is important that this state is reflected by visual appearance.

As a consequence, the property *selected* is added to every object record. We now specify the data types representing lines and captions as follows and note that both types must be extensions of the same base type in order to be members of one and the same data structure.

```
TYPE Object =      POINTER TO ObjectDesc;
  ObjectDesc =     RECORD
                     x, y, w, h, col: INTEGER;
                     selected: BOOLEAN;
                     next: Object
                   END ;

  Line =           POINTER TO LineDesc;
  LineDesc =       RECORD (Object) END ;

  Caption =        POINTER TO CaptionDesc
  CaptionDesc =    RECORD (Object)
                     pos, len: INTEGER
                   END
```

Selection of a single element is typically achieved by pointing at the object with mouse and cursor. Selection of a set of objects is achieved by specifying a rectangular area, implying selection of all objects lying within it. In both cases, the search for selected elements proceeds through the linked list and relies on the position and size stored in each object's descriptor. As a consequence, the rule was adopted that every object not only specify a position through its coordinates *x, y*, but also the rectangular area within which it lies (width *w*, height *h*). It is thus easy to determine whether a given point identifies an object, as well as whether an object lies fully within a rectangular area.

In principle, each caption descriptor carries the sequence of characters (string) representing the caption. The simplest realization would be an array structured field, limiting the length of captions to some fixed, predetermined value. First, this is highly undesirable (although used in early versions of the system). And second, texts carry attributes (color, font). It is therefore best to use a global "scratch text", and to record a caption by the position and length of the string in this immutable text.

A procedure *drawGraphic* to draw all objects of a graphic now assumes the following form:

```
PROCEDURE drawObj(obj: Object);
BEGIN
   IF obj IS Line THEN drawLine(obj(Line))
   ELSIF obj IS Caption THEN drawCaption(obj(Caption))
   ELSE (*other object types, if any*)
   END
END drawObj;

PROCEDRE drawGraphic(first: Object);
   VAR obj: Object;
BEGIN obj := first;
   WHILE obj # NIL DO drawObj(obj); obj := obj.next END
END drawGraphic
```

The two procedures typically are placed in different modules, one containing operations on objects, the other those on graphics. Here the former is the service module, the latter the former's client. Procedures for, e.g, copying elements, or determining whether an object is selectable, follow the same pattern as *drawGraphic*.

This solution has the unpleasant property that all object types are anchored in the base module. If any new types are to be added, the base module has to be modified (and all clients are to be - at least - recompiled). The object-oriented paradigm eliminates this difficulty by inverting the roles of the two modules. It rests on binding the operations pertaining to an object type to each object individually in the form of procedure-typed record fields as shown in the following sample declaration:

```
ObjectDesc =  RECORD
                 x, y, w, h, col: INTEGER; selected: BOOLEAN;
                 draw: PROCEDURE (obj: Object);
                 write:  PROCEDURE (obj: Object; VAR R: Files.Rider);
                 next: Object
              END
```

The procedure *drawGraphic* is now formulated as follows:

```
PROCEDURE drawGraphic(first: Object);
   VAR obj : Object;
BEGIN obj := first;
   WHILE obj 9 NIL DO obj.draw(obj);  obj := obj.next END
END drawGraphic;
```

The individual procedures - in object-oriented terminology called *methods* - are assigned to the record's fields upon its creation. They need no further discrimination of types, as this role is assumed by the assignment of the procedures upon their installation. We note here that the procedure fields are never changed; they assume the role of *constants* rather than variables associated with each object.

This example exhibits in a nutshell the essence of object-oriented programming, *extensibility* as its purpose and the *procedure-typed* record field as the technique.

The given solution, as it stands, has the drawback that each object (instance, variable) contains several procedures (of which three are listed), and therefore leads to a storage requirement that should be avoided. Furthermore, it defines once and for all the number of operations applicable to objects, and also their parameters and result types. A different approach with the same underlying principle removes these drawbacks. It employs a single installed procedure which itself discriminates among the operations according to different types of parameters. The parameters of the preceding solution are merged into a single record called a *message*. The unified procedure is called a *handler*, and messages are typically extensions of a single base type *Msg*.

```
TYPE Msg =       RECORD END;
   DrawMsg =     RECORD (Msg) END;
   WriteMsg =    RECORD (Msg) R: Files.Rider END ;

   ObjectDesc =  RECORD
                    x, y, w, h, col: INTEGER; selected: BOOLEAN;
```

```
                              handle: PROCEDURE (obj: Object; VAR M: Msg);
                              next: Object
                          END ;

    PROCEDURE  Handler (obj: Object; VAR M: Msg);
        (*this procedure is assigned to the handle field of every line object*)
    BEGIN
        IF M IS DrawMsg THEN drawLine(obj(Line))
        ELSIF M IS WriteMsg THEN writeLine(obj(Line), M(WriteMsg).R)
        ELSE ...
        END
    END ;

    PROCEDURE drawGraphic(first: Objec; VAR M: Msg);
        VAR obj: Object;
    BEGIN obj := first;
        WHILE obj 9 NIL DO obj.handle(obj, M);  obj := obj.next END
    END drawGraphics
```

In the present system, a combination of the two schemes presented so far is used. It eliminates the need for individual method fields in each object record as well as the cascaded IF statement for discriminating among the message types. Yet it allows further addition of new methods for later extensions without the need to change the object's declaration. The technique used is to include a single field (called *do*) in each record (analogous to the handler). This field is a pointer to a method record containing the procedures declared for the base type. At least one of them uses a message parameter, i.e. a parameter of record structure that is extensible.

```
  TYPE Method =       POINTER TO MethodDesc;
      Msg =           RECORD END;
      Context =       RECORD END;

      Object =        POINTER TO ObjectDesc;
      ObjectDesc =    RECORD
                          x, y, w, h, col: INTEGER; selected: BOOLEAN;
                          do: Method; next: Object
                      END;

  MethodDesc =        RECORD
                          new: Modules.Command;
                          copy: PROCEDURE (obj, to: Object);
                          draw, handle: PROCEDURE (obj: Object; VAR M: Msg);
                          selectable: PROCEDURE (obj: Object; x, y: INTEGER): BOOLEAN;
                          read: PROCEDURE (obj: Object; VAR R: Files.Rider; VAR C: Context);
                          write: PROCEDURE (obj: Object; cno: SHORTINT;
                              VAR R: Files.Rider; VAR C: Context);
                      END
```

A single method instance is generated when a new object type is created, typically in the initialization sequence of the concerned module. When a new object is created, a pointer to this record is assigned to the do field of the new object descriptor. A call then has the form *obj.do.write(obj, R).* This example exhibits the versatility of Oberon's type extension and procedure variable features very well, and it does so without hiding the data structures involved in a dispensible, built-in run-time mechanism.

The foregoing deliberations suggest the system's modular structure shown in Figure 13.3.:

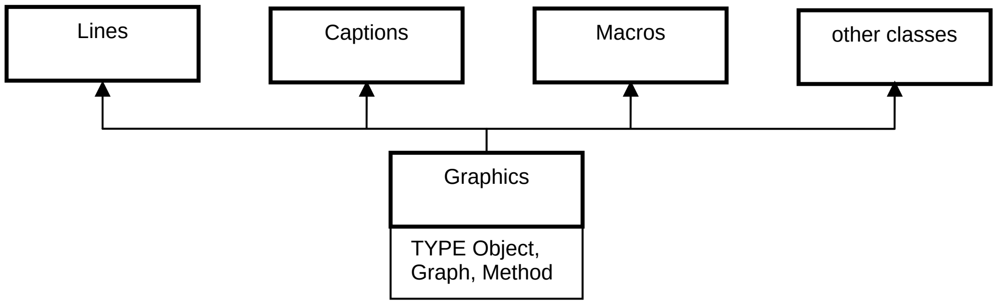


Figure 13.3 Clients of module *Graphics*

The modules in the top row implement the individual object types' methods, and additionally provide commands, in particular *Make* for creating new objects. The base module specifies the base types and procedures operating on graphics as a whole.

Our system, however, deviates from this scheme somewhat for several reasons:

1. Implementation of the few methods requires relatively short programs for the basic objects. Although a sensible modularization is desirable, we wish to avoid an atomization, and therefore merge parts that would result in tiny modules with the base module.
2. The elements of a graphic refer to fonts used in captions and to libraries used in macros. The writing and reading procedures therefore carry a context consisting of fonts and libraries as an additional parameter. Routines for mapping a font (library) to a number according to a given context on output, and a number to a font (library) on input are contained in module *Graphics*.
3. In the design of the Oberon System, a hierarchy of four modules has proven to be most appropriate:

   0. Module with base type handling the abstract data structure.
   1. Module containing procedures for the representation of objects in frames (display handling).
   2. Module containing the primary command interpreter and connecting frames with a viewer.
   3. A command module scanning command lines and invoking the appropriate interpreters.

The module hierarchy of the Graphics System is here shown together with its analogy, with the Text System:

Function	Graphics	Text
3. Command Scanner	Draw	Edit
2. Viewer Handler	MenuViewers	MenuViewers
1. Frame Handler	GraphicFrames	TextFrames
0. Base	Graphics	Texts

As a result, module *Graphics* does not only contain the base type *Object*, but also its extensions *Line* and *Caption* (and *Macro*). Their methods are also defined in *Graphics*, with the exception of drawing methods, which are defined in *GraphicFrames*, because they refer to frames.

So far, we have discussed operations on individual objects and the structure resulting from the desire to be able to add new object types without affecting the base module. We now turn our attention briefly to operations on graphics as a whole. They can be grouped into two kinds, namely operations involving a graphic as a set, and those applying to the selection, i.e. to a subset only.

The former kind consists of procedures Add, which inserts a new object, *Draw*, which traverses the set of objects and invokes their drawing methods, *ThisObj*, which searches for an object at a given position, *SelectObj*, which marks an object to be selected, *SelectArea*, which identifies all objects lying within a given rectangular area and marks them, *Selectable*, a Boolean function, and *Enumerate*, which applies the parametric procedure handle to all objects of a graphic. Furthermore, the procedures *Load, Store, Print,* and *WriteFile* belong to this kind.

The set of operations applying to selected objects only consist of the following procedures: *Deselect, DrawSel* (drawing the selection according to a specified mode), *Change* (changing certain attributes of selected objects like width, font, color), *Move, Copy, CopyOver* (copying from one graphic into another), and finally *Delete*. Also, there exists the important procedure *Open* which creates a new graphic, either loading a graphic stored as a file, or generating an empty graphic.

The declaration of types and procedures that have emerged so far are summarized in the following excerpt of the module's interface definition.

```
DEFINITION Graphics;  (*excerpt without macros*)
  IMPORT Files, Fonts, Texts, Modules, Display;

  CONST NameLen = 32;

  TYPE Graph = POINTER TO GraphDesc;
    Object = POINTER TO ObjectDesc;
    Method = POINTER TO MethodDesc;

    ObjectDesc = RECORD
      x, y, w, h: INTEGER;
      col: BYTE;
      selected, marked: BOOLEAN;
      do: Method
    END ;

    Msg = RECORD END ;
    WidMsg = RECORD (Msg) w: INTEGER END ;
    ColorMsg = RECORD (Msg) col: INTEGER END ;
    FontMsg = RECORD (Msg) fnt: Fonts.Font END ;
    Name = ARRAY NameLen OF CHAR;

    GraphDesc = RECORD sel: Object;
      time: INTEGER
     END ;

    Context = RECORD END ;

    MethodDesc = RECORD
      module, allocator: Name;
      new: Modules.Command;
      copy: PROCEDURE (obj, to: Object);
      draw, change: PROCEDURE (obj: Object; VAR msg: Msg);
      selectable: PROCEDURE (obj: Object; x, y: INTEGER): BOOLEAN;
      raed: PROCEDURE (obj: Object; VAR R: Files.Rider; VAR C: Context);
      write: PROCEDURE (obj: Object; cno: INTEGER; VAR R: Files.Rider; VAR C: Context);
    END ;

    Line = POINTER TO LineDesc;
    LineDesc = RECORD (ObjectDesc) END ;

    Caption = POINTER TO CaptionDesc;
    CaptionDesc = RECORD (ObjectDesc) pos, len: INTEGER END ;

  VAR width, res: INTEGER;
    T: Texts.Text;
    LineMethod, CapMethod, MacMethod: Method;

  PROCEDURE New(obj: Object);
  PROCEDURE Add (G: Graph; obj: Object);
  PROCEDURE Draw (G: Graph; VAR M: Msg);
  PROCEDURE ThisObj (G: Graph; x, y: INTEGER): Object;
  PROCEDURE SelectObj (G: Graph; obj: Object);
  PROCEDURE SelectArea (G: Graph; x0, y0, x1, y1: INTEGER);


  PROCEDURE Deselect (G: Graph);
  PROCEDURE DrawSel (G: Graph; VAR M: Msg);
```

```
    PROCEDURE Change (G: Graph; VAR M: Msg);
    PROCEDURE Move (G: Graph; dx, dy: INTEGER);
    PROCEDURE Copy (Gs, Gd: Graph; dx, dy: INTEGER);
    PROCEDURE Delete (G: Graph);

    PROCEDURE FontNo (VAR W: Files.Rider; VAR C: Context; fnt: Fonts.Font):  INTEGER;
    PROCEDURE WriteObj (VAR W: Files.Rider; cno:  INTEGER; obj: Object);
    PROCEDURE Store (G: Graph; VAR W: Files.Rider);
    PROCEDURE WriteFile (G: Graph; name: ARRAY OF CHAR);
    PROCEDURE Font (VAR R: Files.Rider; VAR C: Context): Fonts.Font;
    PROCEDURE Load (G: Graph; VAR R: Files.Rider);
    PROCEDURE Open (G: Graph; name: ARRAY OF CHAR);
  END Graphics.
```

13.4. Displaying graphics

The base module *Graphics* defines the representation of a set of objects in terms of a data structure. The particulars are hidden and allow the change of structural representation by an exchange of this module without affecting its clients. The problems of displaying a graphic on a screen or a printed page are not handled by this module; they are delegated to the client module *GraphicFrames*, which defines a frame type for graphics which is an extension of *Display.Frame*, just like *TextFrames.Frame* is an extension of *Display.Frame*. In contrast to text frames, however, a graphic instead of a text is associate with it.

```
FrameDesc = RECORD (Display.FrameDesc)
          graph: Graphics.Graph;
          Xg, Yg, X1, Y1, x, y, col: INTEGER;
          marked, ticked: BOOLEAN;
          mark: LocDesc
      END
```

Every frame specifies its coordinates *X, Y* within the display area, its size by the attributes *W* (width) and *H* (height), and its background color *col*. Just as a frame represents a (rectangular) section of the entire screen, it also shows an excerpt of the drawing plane of the graphic. The coordinate origin need coincide with neither the frame origin nor the display origin. The frame's position relative to the graphic plane's origin is recorded in the frame descriptor by the coordinates *Xg, Yg*.

The additional, redundant attributes *x, y, X1, Y1* are given by the following invariants, and they are recorded in order to avoid their frequent recomputation.

$$X1 = X + W, \qquad Y1 = Y + H$$
$$x = X + Xg, \qquad y = Y1 + Yg$$

X and *Y* (and hence also *X1* and *Y1*) are changed when a viewer is modified, i.e. when the frame is moved or resized. *Xg* and *Yg* are changed when the graph's origin is moved within a frame. The meaning of the various values is illustrated in Figure 13.4.

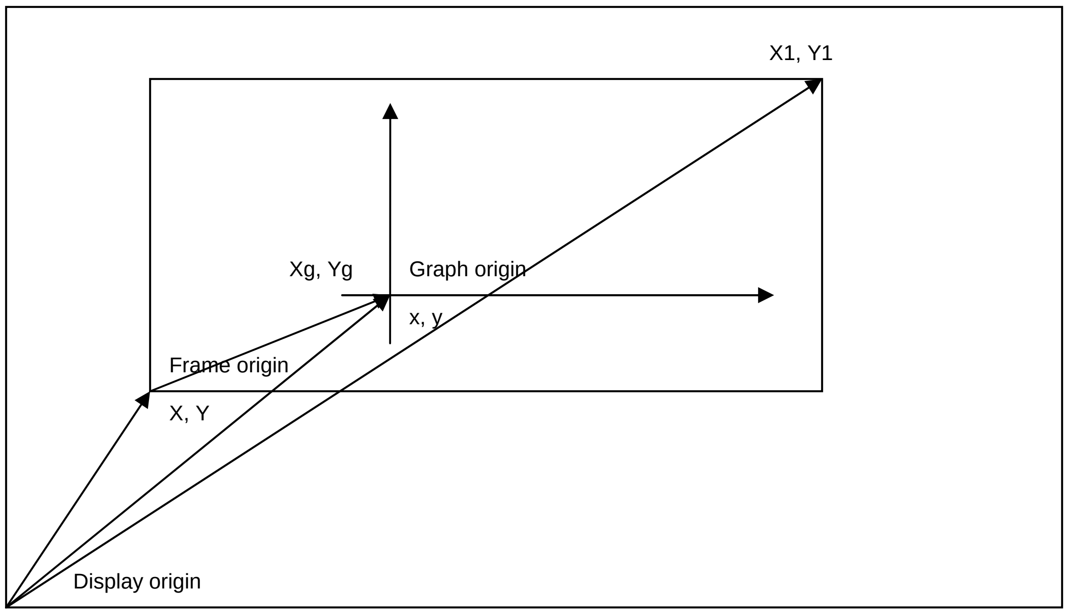


Figure 13.4 Frame and graph coordinates

As a consequence, the display coordinates *u, v* of an object *z* of a graph displayed in a frame *f* are computed as

u = z.x + f.x,   v = z.y + f.y

In order to determine whether an object *z* lies within a frame *f*, the following expression must hold:

(f.X <= u) & (u + z.w <= f.X1) & (f.Y <= v) & (v + z.h <= f.Y1)

The record field *marked* indicates whether or not the frame contains a caret. Its display position is recorded in the field called *mark*. A frame may contain several (secondary) carets; they form a list of location descriptors.

When an object is displayed (drawn), its state must be taken into consideration in order to provide visible user feedback. The manner in which selection is indicated, however, may vary among different object types. This can easily be realized, because every object (type) is associated with an individual drawing procedure. The following visualizations of selection have been chosen:

Selected lines are shown in a grey tone (raster pattern).
Selected captions are shown with "inverse video".

Change of state is a relatively frequent operation, and if possible a complete repainting of the involved objects should be avoided for reasons of efficiency. Therefore, procedures for drawing an object are given a mode parameter, in addition to the obvious object and frame parameters. The parameters are combined into the message record of type *DrawMsg*.

```
DrawMsg =    RECORD (Graphics.Msg)
                 f: Frame;
                 mode, x, y, col: INTEGER
             END
```

The meaning of the mode parameter's four possible values are the following:

mode = 0:   draw object according to its state,
mode = 1:   draw reflecting a transition from normal to selected state,
mode = 2:   draw reflecting a transition from selected to normal state,
mode = 3:   erase.

In the case of captions, for instance, the transitions are indicated by simply inverting the rectangular area covered by the caption. No rewriting of the captions' character patterns is required.

A mode parameter is also necessary for reflecting object deletion. First, the selected objects are drawn with *mode* indicating erasure. Only afterwards are they removed from the graphic's linked list.

Furthermore, the message parameter of the drawing procedure contains two offsets *x* and *y*. They are added to the object's coordinates, and their significance will become apparent in connection with macros. The same holds for the color parameter.

The drawing procedures are fairly straight-forward and use the four basic raster operations of module *Display*. The only complication arises from the need to clip the drawing at the frame boundaries. In the case of captions, a character is drawn only if it fits into the frame in its entirety. The raster operations do not test (again) whether the indicated position is valid.

At this point we recall that copies of a viewer (and its frames) can be generated by the *System.Copy* command. Such copies display the same graphic, but possibly different excerpts of them. When a graphic is changed by an insertion, deletion, or any other operation, at a place that is visible in several frames, all affected views must reflect the change. A direct call to a drawing procedure indicating a frame and the change does therefore not suffice. Here again, the object-oriented style solves the problem neatly: In place of a direct call a message is broadcast to all frames, the message specifying the nature of the required updates.

The broadcast is performed by the general procedure *Viewers.Broadcast(M)*. It invokes the handlers of all viewers with the parameter *M*. The viewer handlers either interpret the message or propagate it to the handlers of their subframes. Procedure *obj.handle* is called with a control message as parameter when pointing at the object and clicking the middle mouse button. This allows control to be passed to the handler of an individual object.

The definition of module *GraphicFrames* is summarized by the following interface:

```
DEFINITION GraphicFrames;
    IMPORT Display, Graphics;

    TYPE Frame = POINTER TO FrameDesc;
        Location = POINTER TO LocDesc;

        LocDesc = RECORD
            x, y: INTEGER;
            next: Location
        END ;

        FrameDesc = RECORD (Display.FrameDesc)
            graph: Graphics.Graph;
            Xg, Yg, X1, Y1, x, y, col: INTEGER;
            marked, ticked: BOOLEAN;
            mark: LocDesc
        END ;

        (*mode = 0: draw according to selected, 1: normal -> selected, 2: selected -> normal, 3: erase*)

        DrawMsg = RECORD (Graphics.Msg)
            f: Frame;
            x, y, col, mode: INTEGER
        END ;

    PROCEDURE Restore (F: Frame);
    PROCEDURE Focus (): Frame;
    PROCEDURE Selected (): Frame;
    PROCEDURE This(x, y: INTEGER): Frame;
    PROCEDURE Draw (F: Frame);
    PROCEDURE Erase (F: Frame);
    PROCEDURE DrawObj (F: Frame; obj: Graphics.Object);
    PROCEDURE EraseObj (F: Frame; obj: Graphics.Object);
    PROCEDURE Change (F: Frame; VAR msg: Graphics.Msg);
    PROCEDURE Defocus (F: Frame);
    PROCEDURE Deselect (F: Frame);
    PROCEDURE Macro (VAR Lname, Mname: ARRAY OF CHAR);
    PROCEDURE Open (G: Frame; graph: Graphics.Graph);
END GraphicFrames.
```

Focus and *Selected* identify the graphic frame containing the caret, or containing the latest selection. *Draw, Erase,* and *Handle* apply to the selection of the specified frame's graphic. And *Open* generates a frame displaying the specified graphic.

13.5. The user interface

Although the display is the prime constituent of the interface between the computer and its user, we chose the title of this chapter for a presentation primarily focussed on the computer's input, i.e. on its actions instigated by the user's handling of keyboard and mouse, the editing operations. The design of the user interface plays a decisive role in a system's acceptance by users. There is no fixed set of rules which determine the optimal choice of an interface. Many issues are a matter of subjective judgement, and all too often convention is being mixed up with convenience. Nevertheless, a few criteria have emerged as fairly generally accepted.

We base our discussion on the premise that input is provided by a keyboard and a mouse, and that keyboard input is essentially to be reserved for textual input. The critical issue is that a mouse - apart from providing a cursor position - allows to signal actions by the state of its keys. Typically, there are far more actions than there are keys. Some mice feature a single key only, a situation that we deem highly unfortunate. There are, however, several ways to "enrich" key states:

1. Position. Key states are interpreted depending on the current position of the mouse represented by the cursor. Typically, interpretation occurs by the handler installed in the viewer covering the cursor position, and different handlers are associated with different viewer types. The handler chosen for interpretation may even be associated with an individual (graphic) object and depend on that object's type.
2. Multiple clicks. Interpretation may depend on the number of repeated clicks (of the same key), and/or on the duration of clicks.
3. Interclicks. Interpretation may depend on the combination of keys depressed until the last one is released. This method is obviously inapplicable for single-key mice.

Apart from position dependence, we have quite successfully used interclicks. A ground rule to be observed is that frequent actions should be triggered by single-key clicks, and only variants of them should be signalled by interclicks. The essential art is to avoid overloading this method.

Less frequent operations may as well be triggered by textual commands, i.e. by pointing at the command word and clicking the middle button. Even for this kind of activation, Oberon offers two variations:

1. The command is listed in a menu (title bar). This solution is favoured when the respective viewer is itself a parameter to the command, and it is recommended when the command is reasonably frequent, because the necessary mouse movement is relatively short.
2. The command lies elsewhere, typically in a viewer containing a tool text.

Lastly, we note that any package such as *Draw* is integrated within an entire system together with other packages. Hence it is important that the rules governing the user interfaces of the various packages do not differ unnecessarily, but that they display common ground rules and a common design "philosophy". *Draw*'s conventions were, as far as possible and sensible, adapted to those of the text system. The right key serves for selection, the left for setting the caret, and the middle key for activating general commands, in this case moving and copying the entire graphic. Inherently, drawing involves certain commands that cannot be dealt with in the same way as for texts. A character is created by typing on the keyboard; a line is created by dragging the mouse while holding the left key. Interclicks left-middle and right-middle are treated in the same way as in the text system (copying a caption from the selection to the caret), and this is not surprising, because text and graphics are properly integrated, i.e. captions can be copied from texts into graphics and vice-versa.

Using different conventions depending on whether the command was activated by pointing at the caption within a text frame or within a graphics frame would be confusing indeed.

13.6. Macros

For many applications it is indispensible that certain sets of objects may be named and used as objects themselves. Such a named subgraph is called a *macro*. A macro thus closely mirrors the sequence of statements in a program text that is given a name and can be referenced from within other statements: the procedure. The notion of a graphic object becomes recursive, too. The facility of recursive objects is so fundamental that it was incorporated in the base module *Graphics* as the third class of objects.

Its representation is straight-forward: in addition to the attributes common to all objects, a field is provided storing the head of the list of elements which constitute the macro. In the present system, a special node is introduced representing the head of the element list. It is of type *MacHeadDesc* and carries also the name of the macro and the width and height of the rectangle covering all elements. These values serve to speed up the selection process, avoiding their recomputation by scanning the entire element list.

The recursive nature of macros manifests itself in recursive calls of display procedures. In order to draw a macro, drawing procedures of the macro's element types are called (which may be macros again). The coordinates of the macro are added to the coordinates of each element, which function as offsets. The color value of the macro, also a field of the parameter of type *DrawMsg*, overrides the colors of the elements. This implies that macros always appear monochrome.

An application of the macro facility is the design of schematics of electronic circuits. Circuit components correspond to macros. Most components are represented by a rectangular frame and by labelled connectors (pins). Some of the most elementary components, such as gates, diodes, transistors, resistors, and capacitors are represented by standardized symbols. Such symbols, which may be regarded as forming an alphabet of electronic circuit diagrams, are appropriately provided in the form of a special font, i.e. a collection of raster patterns. Three such macros are shown in Figure 13.5, together with the components from which they are assembled. The definitions of the data types involved are:

```
Macro =          POINTER TO MacroDesc;
MacroDesc =      RECORD (ObjectDesc) mac: MacHead END ;

MacHead =        POINTER TO MacHeadDesc;
MacHeadDesc =    RECORD name: Name;
                     w, h: INTEGER; lib: Library
                 END ;

Library =        POINTER TO LibraryDesc;
LibraryDesc =    RECORD name: Name END
```

Procedure *DrawMac(mh, M)* displays the macro with head *mh* according to the draw message parameter *M* which specifies a frame, a position within the frame, a display mode, and an overriding color.

In the great majority of applications, macros are not created by their user, but are rather provided from another source, in the case of electronic circuits typically by the manufacturer of the components represented by the macros. As a consequence, macros are taken from a collection (inappropriately) called a *library*. In our system, a macro is picked from such a collection by the command *Draw.Macro* with a library name and a macro name as parameters. It inserts the specified macro at the place of the caret by calling *GraphicFrames.Macro*, which in turn calls *Graphics.Add.*

At last, we mention that selection of a macro is visualized by covering with a dot pattern the entire rectangular area occupied by the macro. This emphasizes the fact that the macro constitutes an object as a whole.

The design of new macros is a relatively rare activity. Macros are used rather like characters of a font; the design of new macros and fonts is left to the specialist. Nevertheless, it was decided to incorporate the ingredients necessary for macro design in the basic system. They consist of a few procedures only which are used by a tool module called *MacroTool* (see Section 16.3).

MakeMac integrates all elements lying within a specified rectangular area into a new macro. *OpenMac* reverses this process by disintegrating the macro into its parts. *InsertMac* inserts a specified macro into a library. *NewLib* creates a new, empty library, and *StoreLib* generates a library file containing all macros currently loaded into the specified library. The details of these operations may be examined in the program listings provided later in this Chapter. Summarizing, the following procedures are exported from module *Graphics* related to handling macros:

```
PROCEDURE GetLib(name: ARRAY OF CHAR; replace: BOOLEAN; VAR Lib: Library);
PROCEDURE ThisMac(L: Library; Mname: ARRAY OF CHAR): MacHead;
PROCEDURE DrawMac(mh: MacHead; VAR M: Msg);
```

and the following are added for creating new macros and libraries:

```
PROCEDURE NewLib(Lname: ARRAY OF CHAR): Library;
PROCEDURE StoreLib(L: Library; Fname: ARRAY OF CHAR);
PROCEDURE RemoveLibraries;
PROCEDURE OpenMac(mh: MacHead; G: Graph; x, y: INTEGER);
PROCEDURE MakeMac(G: Graph; x, y, w, h: INTEGER; Mname: ARRAY OF CHAR): MacHead;
PROCEDURE InsertMac(mh: MacHead; L: Library; VAR new: BOOLEAN);
```

13. 7. Object classes

Although surprisingly many applications can be covered satisfactorily with the few types of objects and the few facilities described so far, it is nevertheless expected that a modern graphics system allow the addition of further types of objects. The emphasis lies here on the word addition instead of change. New facilities are to be providable by the inclusion of new modules without requiring any kind of adjustment, not even recompilation of the existing modules. In practice, their source code would quite likely not be available. It is the triumph of the object-oriented programming technique that this is elegantly possible. The means are the extensible record type and the procedure variable, features of the programming language, and the possibility to load modules on demand from statements within a program, a facility provided by the operating environment.

We call, informally, any extension of the type *Object* a *class*. Hence, the types *Line, Caption,* and *Macro* constitute classes. Additional classes can be defined in other modules importing the type *Object*. In every such case, a set of methods must be declared and assigned to a variable of type *MethodDesc*. They form a so-called *method suite*. Every such module must also contain a procedure, typically a command, to generate a new instance of the new class. This command, likely to be called *Make*, assigns the method suite to the *do* field of the new object.

This successful decoupling of additions from the system's base suffices, almost. Only one further link is unavoidable: When a new graphic, containing objects of a class not defined in the system's core, is loaded from a file, then that class must be identified, the corresponding module with its handlers must be loaded - this is called *dynamic loading* - and the object must be generated (allocated). Because the object in question does not already exist at the time when reading the object's attribute values, the generating procedure cannot possibly be installed in the very same object, i.e. it cannot be a member of the method suite. We have chosen the following solution to this problem:

1. Every new class is implemented in the form of a module, and every class is identified by the module name. Every such module contains a command whose effect is to allocate an object of the class, to assign the message suite to it, and to assign the object to the global variable *Graphics.new*.

2. When a graphics file is read, the class of each object is identified and a call to the respective module's allocation procedure delivers the desired object. The call consists of two parts: a call to *Modules.ThisMod*, which may cause the loading of the respctive class module *M*, and a call of

Modules.ThisCommand. Then the data of the base type *Object* are read, and lastly the data of the extension are read by a call to the class method *read*.

The following may serve as a template for any module defining a new object class *X*. Two examples are given in Section 13.9, namey *Rectangles* and *Curves*.

```
MODULE Xs;
  IMPORT Files, Oberon, Graphics, GraphicFrames;

  TYPE X* = POINTER TO XDesc;
    XDesc = RECORD (Graphics.ObjectDesc) (*additional data fields*) END ;

  VAR method: Graphics.Method;

  PROCEDURE New*;
    VAR x: X;
  BEGIN NEW(x); x.do := method; Graphics.new := x
  END New;

  PROCEDURE* Copy(obj, to: Graphics.Object);
  BEGIN to(X)^ := obj(X)^
  END Copy;

  PROCEDURE* Draw(obj: Graphics.Object; VAR msg: Graphics.Msg);
  BEGIN ...
  END Draw;

  PROCEDURE* Selectable(obj: Graphics.Object; x, y: INTEGER): BOOLEAN;
  BEGIN ...
  END Selectable;

  PROCEDURE* Change(obj: Graphics.Object; VAR msg: Graphics.Msg);
  BEGIN
    IF msg IS Graphics.ColorMsg THEN obj.col := msg(Graphics.ColorMsg).col
    ELSIF msg IS ... THEN ...
    END
  END Handle;

  PROCEDURE* Read(obj: Graphics.Object; VAR W: Files.Rider; VAR C: Context);
  BEGIN (*read X-specific data*)
  END Write;

  PROCEDURE* Write(obj: Graphics.Object; cno: SHORTINT;
    VAR W: Files.Rider; VAR C: Context);
  BEGIN Graphics.WriteObj(W, cno, obj); (*write X-specific data*)
  END Write;

  PROCEDURE Make*;   (*command*)
    VAR x: X;  F: GraphicFrames.Frame;
  BEGIN F := GraphicFrames.Focus();
    IF F # NIL THEN
      GraphicFrames.Deselect(F);
      NEW(x); x.x := F.mark.x - F.x; x.y := F.mark.y - F.y; x.w := ... ; x.h := ... ;
      x.col := Oberon.CurCol; x.do := method;
      GraphicFrames.Defocus(F); Graphics.Add(F.graph, x); GraphicFrames.DrawObj(F, x)
    END
  END Make;

BEGIN NEW(method); method.module := "Xs"; method.allocator := "New";
  method.copy := Copy; method.draw := Draw; method.selectable := Selectable;
  method.handle := Handle; method.read := Read; method.write := Write; method.print := Print
END Xs.
```

We wish to point out that also the macro and library facilities are capable of integrating objects of new classes, i.e. of types not occurring in the declarations of macro and library facilities. The complete interface definition of module *Graphics* is obtained from its excerpt given in Sect. 13.3, augmented by the declarations of types and procedures in Sect. 13.6. and 13.7.

13.8. The implementation

13.8.1. Module Draw

Module *Draw* is a typical command module whose exported procedures are listed in a tool text. Its task is to scan the text containing the command for parameters, to check their validity, and to activate the corresponding procedures, which primarily are contained in modules *Graphics* and *GraphicFrames.* The most prominent among them is the *Open* command. It generates a new viewer containing two frames, namely a text frame serving as menu, and a graphic frame.

We emphasize at this point that graphic frames may be opened and manipulated also by other modules apart from *Draw*. In particular, document editors that integrate texts and graphics - and perhaps also other entities - would refer to *Graphics* and *GraphicFrames* directly, but not make use of *Draw* which, as a tool module, should not have client modules.

```
DEFINITION Draw;
    PROCEDURE Open;
    PROCEDURE Delete;
    PROCEDURE SetWidth;
    PROCEDURE ChangeColor;
    PROCEDURE Store;
    PROCEDURE Macro;

    PROCEDURE OpenMacro;
    PROCEDURE MakeMacro;
    PROCEDURE LoadLibrary;
END Draw.
```

13.8.2. Module *GraphicFrames*

Module *GraphicFrames* contains all routines concerned with displaying, visualizing graphic frames and their contents, i.e. graphics. It also contains the routines for creating new objects of the base classes, i.e. lines, captions, and macros. And most importantly, it specifies the appropriate frame handler which interprets input actions and thereby defines the user interface. The handler discriminates among the following message types:

1. Update messages. According to the *id* field of the message record, either a specific object or the entire selection of a graphic are drawn according to a mode. The case *id = 0* signifies a restoration of the entire frame including all objects of the graphic.

2. Selection, focus, and position queries. They serve for the identification of the graphic frame containing the latest selection, containing the caret (mark) or the indicated position. In order to identify the latest selection, the time is recorded in the graph descriptor whenever a new selection is made or when new objects are inserted.

3. Input messages. They originate from the central loop of module *Oberon* and indicate either a mouse action (track message) or a keyboard event (consume message).

4. Control messages from *Oberon*. They indicate that all marks (selection, caret, star) are to be removed (neutralize), or that the focus has to be relinquished (defocus).

5. Selection and copy messages from *Oberon*. They constitute the interface between the graphics and the text system, and make possible identification and copying of captions between graphic and text frames.

6. Modify messages from *MenuViewers*. They indicate that a frame has to be adjusted in size and position because a neighbouring viewer has been reshaped, or because its own viewer has been repositioned

7. Display messages. They originate from procedure *InsertChar* and handle the displaying of single characters when a caption is composed (see below).

The frame handler receiving a consume message interprets the request through procedure *InsertChar*, and receiving a track message through procedure *Edit*. If no mouse key is depressed, the cursor is simply drawn, and thereby the mouse is tracked. Instead of the regular arrow, a crosshair is used as cursor pattern. Thereby immediate visual feedback is provided to indicate that now mouse actions are interpreted by the graphics handler (instead of, e.g., a text handler). Such feedback is helpful when graphic frames appear not only in a menuviewer, but as subframes of a more highly structured document frame.

Procedure *Edit* first tracks the mouse while recording further key activities (interclicks) until all keys are released. The subsequent action is determined by the perceived key clicks. The actions are (the second key denotes the interclick):

keys = left	set caret, if mouse was not moved, otherwise draw new line,
keys = left, middle	copy text selection to caret position
keys = left, right	set secondary caret (mark)
keys = middle	move selection
keys = middle, left	copy selection
keys = middle, right	shift origin of graph
keys = right	select (either object, or objects in area)
keys = right, middle	copy selected text to caret position

When copying or moving a set of selected objects, it must be distinguished between the cases where the source and the destination graphics are the same or are distinct. In the former case, source and destination positions may lie in the same or in different frames.

Procedure *InsertChar* handles the creation of new captions. The actual character string is appended to the global text *T*, and the new object records its position within *T* and its length.

A complication arises because the input process consists of as many user actions as there are characters, and because other actions may possibly intervene between the typing. It is therefore unavoidable to record an insertion state, which is embodied by the global variable *newcap*. When a character is typed, and *newcap = NIL*, then a new caption is created consisting of the single typed character. Subsequent typing results in appending characters to the string (and *newcap*). The variable is reset to NIL, when the caret is repositioned. The BS character is interpreted as a backspace by procedure *DeleteChar*.

Since the caption being generated may be visible simultaneously in several frames, its display must be handled by a message. For this reason, the special message *DispMsg* is introduced, and as a result, the process of character insertion turns out to be a rather complex action. To avoid even further complexity, the restriction is adopted that all characters of a caption must use the same attributes (font, color).

The definition of the interface of *GraphicFrames* is listed in Section 13.3.

13.8.3. Module *Graphics*

The preceding presentations of the interface definitions have explained the framework of the graphics system and set the goals for their implementation. We recall that the core module *Graphics* handles the data structures representing sets of objects without reliance on the specifications of individual objects. Even the structural aspects of the object sets are not fixed by the interface. Several solutions, and hence several implementations are imaginable.

Here we present the simplest solution for representing an abstract, unordered set: the linear, linked list. It is embodied in the object record's additional, hidden field *next*. Consequently, a graphic is represented by the head of the list. The type *GraphDesc* contains the hidden field *first* (see listing of *Graphics*). In addition, the descriptor contains the exported field *sel* denoting a selected element, and the field *time* indicating the time of its selection. The latter is used to determine the most recent selection in various viewers.

Additional data structures become necessary through the presence of macros and classes. Macros are represented by the list of their elements, like graphics. Their header is of type *MacHeadDesc* in analogy to *GraphDesc*. In addition to a macro's name, width, and height, it contains the field *first*, pointing to the list's first element, and the field *lib*, referring to the library from which the macro stems.

A library descriptor is similarly structured: In addition to its name, the field *first* points to the list of elements (macros) of the library, which are themselves linked through the field *next*. Fig. 13.6. shows the data structure containing two libraries. It is anchored in the global variable *firstLib*.

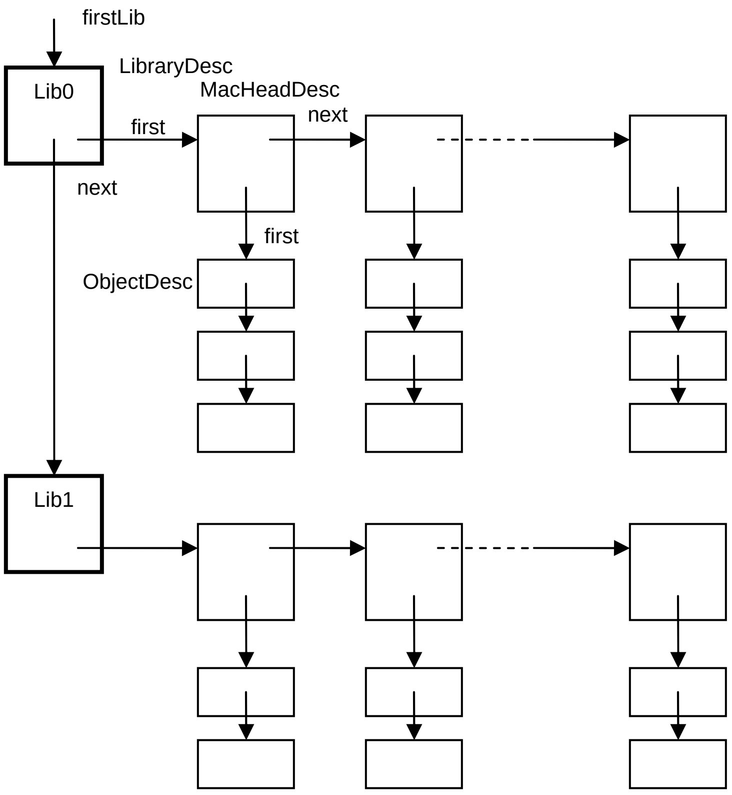


Fig. 13.6 Data structure for two libraries, each with three macros

Libraries are permanently stored as files. It is evidently unacceptable that file access be required upon every reference to a macro, e.g. each time a macro is redrawn. Therefore a library is loaded into primary store, when one of its elements is referenced for the first time. Procedure *ThisMac* searches the data structure representing the specified library and locates the header of the requested macro.

We emphasize that the structures employed for macro and library representation remain hidden from clients, just like the structure of graphics remains hidden within module *Graphics*. Thus, none of the linkage fields of records (*first, next, sel*) are exported from the base module. This measure retains the possibility to change the structural design decisions without affecting the client modules. But partly it is also responsible for the necessity to include macros in the base module.

A large fraction of module *Graphics* is taken up by procedures for reading and writing files representing graphics and libraries. They convert their internal data structure into a sequential form and vice-versa. This would be a rather trivial task, were it not for the presence of pointers referring to macros and classes. These pointers must be converted into descriptions that are position-independent, such as names. The same problem is posed by fonts (which are also represented by pointers).

Evidently, the replacement of every pointer by an explicit name would be an uneconomical solution with respect to storage space as well as speed of reading and writing. Therefore, pointers to fonts and libraries - themselves represented as files - are replaced by indices to font and library dictionaries. These dictionaries establish a context and are constructed while a file is read. They are used only during this process and hence are local to procedure *Load* (or *Open*). For classes, a dictionary listing the respective allocation procedures is constructed in order to avoid repeated calls to determine the pertinent allocator.

When a graphics file is generated by procedure *Store*, local dictionaries are constructed of fonts, libraries, and classes of objects that have been written onto the file. Upon encountering a caption, a macro, or any element whose font, library, or class is not contained in the respective dictionary, a pair consisting of index and name is emitted to the file, thereby assigning a number to each name. These pairs are interspersed within the sequence of object descriptions.

When the graphic file is read, these pairs trigger insertion of the font, library, or class in the respective dictionary, whereby the name is converted into a pointer to the entity, which is obtained by a loading process embodied by procedures *Fonts.This, GetLib,* and *GetClass*. Both the *Load* and *Store* procedures traverse the file only once. The files are self-contained in the sense that all external quantities are represented by their names. The format of a graphics file is defined in Extended BNF syntax as follows:

```
file =     tag stretch.
stretch =  {item} 255.
item =     0 0 fontno fontname | 0 1 libno libname | 0 2 classno classname allocname |
           1 data |  2 data fontno string |  3 data libno macname |  classno data extension.
data =     x  y  w  h  color.
```

All class numbers are at least 4; the values 1, 2, and 3 are assigned to lines, captions, and macros. *x, y, w, h* are two-byte integer attributes of the base type *Object*. The attribute *color* takes a single byte. The first byte of an item being 0 signifies that the item is an identification of a new font, library, or class. If the second byte is 0, a new font is announced, if 1 a new library, and if 2 a new class of elements.

The same procedures are used for loading and storing a library file. In fact, *Load* and *Store* read and write a file stretch representing a sequence of elements which is terminated by a special value (255). In a library file each macro corresponds to a stretch, and the terminator is followed by values specifying the macro's overall width, height, and its name. The structure of library files is defined by the following syntax:

```
libfile =  libtag {macro}.
macro =    stretch w h name.
```

The first byte of each element is a class number within the context of the file and identifies the class to which the element belongs. An object of the given class is allocated by calling the class' allocation procedure, which is obtained from the class dictionary in the given context. The class number is used as dictionary index. The presence of the required allocation procedure in the dictionary is guaranteed by the fact that a corresponding index/name pair had preceded the element in the file.

The encounter of such a pair triggers the loading of the module specifying the class and its methods. The name of the pair consists of two parts: the first specifies the module in which the class is defined, and it is taken as the parameter of the call to the loader (see procedure *GetClass*). The second part is the name of the relevant allocation procedure which returns a fresh object to variable *Graphics.new*. Thereafter, the data defined in the base type *Object* are read.

Data belonging to an extension follow those of the base type, and they are read by the extension's *read* method. This part must always be headed by a byte specifying the number of bytes which follow. This information is used in the case where a requested module is not present; it indicates the number of bytes to be skipped in order to continue reading further elements.

A last noteworthy detail concerns the *Move* operation which appears as surprisingly complicated, particularly in comparison with the related copy operation. The reason is our deviation from the principle that a graphics editor must refrain from an interpretation of drawings. Responsible for this deviation was the circumstance that the editor was at first primarily used for the preparation of circuit diagrams. They suggested the view that adjoining, perpendicular lines be connected. Consequently, the horizontal or vertical displacement of a line was to preserve connections. Procedure *Move* must therefore identify all connected lines, and subsequently extend or shorten them.

The definition of the interface of *Graphics* is listed in Section 13.3.

13.9. Rectangles and curves

13.9.1. Rectangles

In this section, we present two extensions of the basic graphics system which introduce new classes of objects. The first implements rectangles which are typically used for framing a set of objects. They are, for example, used in the representation of electronic components (macros, see Fig. 13.2). Their implementation follows the scheme presented at the end of chapter 13.7 and is reasonably straight-forward, considering that each rectangle merely consists of four lines. Additionally, a background raster may be specified.

One of the design decisions occurring for every new class concerns the way to display the selection. In this case we chose, in contrast to the cases of captions and macros, not inverse video, but a small square dot in the lower right corner of the rectangle. The data type *Rectangle* contains one additional field: *lw* indicates the line width.

In spite of the simplicity of the notion of rectangles, their drawing method is more complex than might be expected. The reason is that drawing methods are responsible for appropriate clipping at frame boundaries. In this case, some of the component lines may have to be shortened, and some may disappear altogether.

Procedure *Handle* provides an example of a receiver of a control message. It is activated as soon as the middle mouse button is pressed, in contrast to other actions, which are initiated after the release of all buttons. Therefore, this message allows for the implementation of actions under control of individual handlers interpreting further mouse movements. In this example, the action serves to change the size of the rectangle, namely by moving its lower left corner.

```
DEFINITION Rectangles;
   TYPE Rectangle = POINTER TO RectDesc;

      RectDesc = RECORD (Graphics.ObjectDesc)
          lw: INTEGER
         END ;

   VAR method: Graphics.Method;

   PROCEDURE New;
   PROCEDURE Make;
END Rectangles.
```

13.9.2. Oblique lines and circles

The second extension to be presented is module *Curves*. It introduces two new kinds of objects: lines which are not necessarily horizontal or vertical, and circles. All are considered to be variants of the same type *Curve*, the variant being specified by the field *kind* of the object record. Selection is indicated by a small rectangle at the end of a line and at the lowest point of a circle.

In order to avoid computations involving floating-point numbers and to increase efficiency, Bresenham algorithms are used. The algorithm for a line defined by $bx - ay = 0$ (for $b \leq a$) is given by the following statements:

```
x := 0; y := 0; h := (b – a) DIV 2;
WHILE x <= a DO Dot(x, y);
  IF h <= 0 THEN INC(h, b) ELSE INC(h, b-a); INC(y) END ;
```

```
   INC(x)
END
```

The Bresenham algorithm for a circle given by the equation $x^2 + y^2 = r^2$ is:

```
x := r; y := 0; h := 1-r;
WHILE y <= x DO Dot(x, y);
   IF h < 0 THEN INC(h, 2*y + 3) ELSE INC(h, 2*(y-x)+5); DEC(x) END ;
   INC(y)
END
```

```
DEFINITION Curves;
   TYPE Curve = POINTER TO CurveDesc;

        CurveDesc = RECORD (Graphics.ObjectDesc)
                kind, lw: INTEGER
             END ;

   (*kind: 0 = up-line, 1 = down-line, 2 = circle*)

   VAR method: Graphics.Method;
   PROCEDURE MakeLine;
   PROCEDURE MakeCircle*;
END Curves.
```

14 Building and maintenance tools

14.1. The Startup Process

An aspect usually given little attention in system descriptions is the process of how a system is started. Its choice, however, is itself an interesting and far from trivial design consideration and will be described here in some detail. Moreover, it directly determines the steps in which a system is developed from scratch, mirroring the steps in which it builds itself up from a bare store to an operating body.

The startup process typically proceeds in several stages, each of them bringing further facilities into play, raising the system to a higher level towards completion. The term for this strategy is *boot strapping* or, in modern computer jargon, *booting*.

Stage 0 is initiated when power is switched on or when the reset button is pressed and released. To be precise, power-on issues a reset signal to all parts of the computer and holds it for a certain time. Pushing the reset button therefore appears like a power-on without power having been switched off. Release of the reset signal triggers the built-in FPGA hardware to load a short *configuration bit-stream* from a ROM residing on the Spartan board, called the *platform flash*, into a BRAM within the FPGA. This program is called *boot loader*. Being stored in a ROM, it is always present. The BRAM is address-mapped onto an upper part of the address space, and the RISC processor starts execution at this address.

In Stage 1 the boot loader loads the *inner core*, which consists of modules *Kernel, FileDir, Files,* and *Modules.* The loader first inspects the link register. If its value is 0, a cold start is indicated. (If the value of the link register is not 0, this signals an abort caused by pressing button 3 on the board. Then loading is skipped and control is immediately returned to the Oberon command loop). The disk (SD-card, SPI) is initialized.

The boot loader terminates with a branch to location 0, which transfers control to the just loaded module *Modules*, the regular loader.

Stage 2 starts with the initialization body of module *Modules* which calls the bodies of *Kernel, FileDir* and *Files*, establishing a working file system. Then it calls itself, requesting to load the central module *Oberon*. This implicitly causes the loading of its own imports, namely *Input, Display, Viewers, Fonts,* and *Texts,* establishing a working viewer and text system.

This loading of the *outer core* must be interpreted as the continuation of the loading of the inner core. To allow proper continuation, the boot loader has deposited the following data in fixed locations:

0	A branch instruction to the initializing body of module *Modules*
12	The limit of available memory
16	The address of the end of the module space loaded
20	The current root of the links of loaded modules
24	The current limit of the module area

In Stage 3, *Oberon* calls the loader to load the tool module *System*, and with it its imports *MenuViewers* and *TextFrames*. The initialization of *System* causes the opening of the viewers for the system tool and the system log. Control then returns to *Oberon* and its central loop for polling input events. Normal operation begins. The booting process is summarized in Figure 14.1.

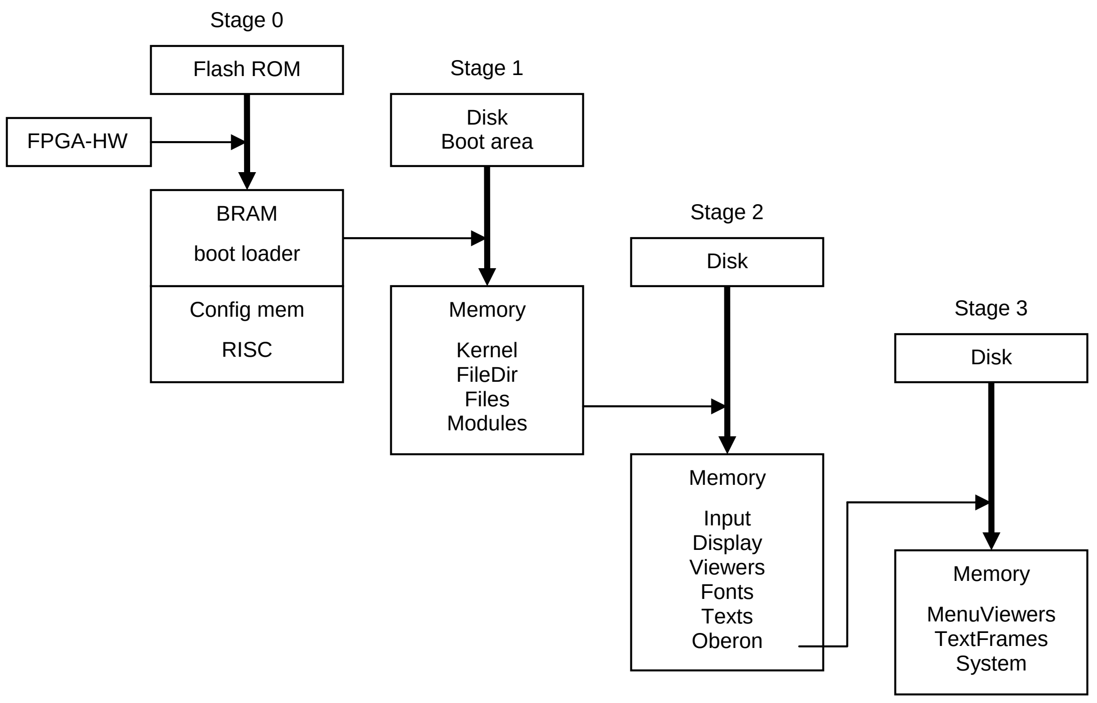


Figure 14.1 The four stages of the booting process

This describes the normal case of startup. But, how did the boot loader ever get into the platform-flash, and how did the inner core ever get into the boot area of the disk, and how did the files of the outer core get into the file store? In fact, how did the file store get initialized? This is described in the following section on building tools.

Precisely to solve this problem, the boot loader has been provided with a second source of the boot data. Instead of from the disk, it may be fetched over a data link, in this case the RS-232 data line. This choice is set by switch 0.

0	load from the "boot track" of the disk (sectors 2 - 63)
1	load from the RS-232 line (or a network, if available)

In case 1, the data stream originates at a host computer, on which presumably the boot file had been generated or even the entire system had been built.

In order to keep the boot loader as simple as possible - remember that it is placed in a small flash memory on every workstation and therefore cannot be changed without a special effort - the format of the byte stream representing the inner core must be simple. We have chosen the following structure, which had never to be changed during the entire development effort of the Oberon System because of both its simplicity and generality:

```
BootFile    = {block}.
block       = size address {byte}.   (size and address are words)
```

The address of the last block, distinguished by *size* = 0, is interpreted as the address of the starting point of Stage 2.

In this step, a module called *Oberon0* is used as the top module, rather than *Modules*. This module communicates with the host computer via the RS-232 line and in addition features various inspection tools. In particular it contains a command copying the just loaded inner core into the disk (see also Section 14.2).

Still, how did the hardware configuration data and the boot loader get into the Flash ROM? This step requires the help of proprietary tools of the FPGA manufacturer. Regrettably, their incantation ceremony typically is rather complex.

After all necessary Verilog modules have been synthesized, the result is the configuration file *RISCTop.bit.* The necessary source files are

RISCTop.v, RISC.v, Multiplier.v, Divider.v, FPAdder.v. FP.Multiplier.v, FP.Divider.v, dbram32.v
RS232R.v, RS232T.v, SPI.v, XGS.v, PS2.v, RISC.ucf

Thereafter, the boot loader is compiled and, together with the result of the configuration of the RISC hardware, loaded into the configuration memory of the FPGA. This Stage 0 is partly done with proprietary software (dependant on the specific FPGA) and is described in a separate installation guide.

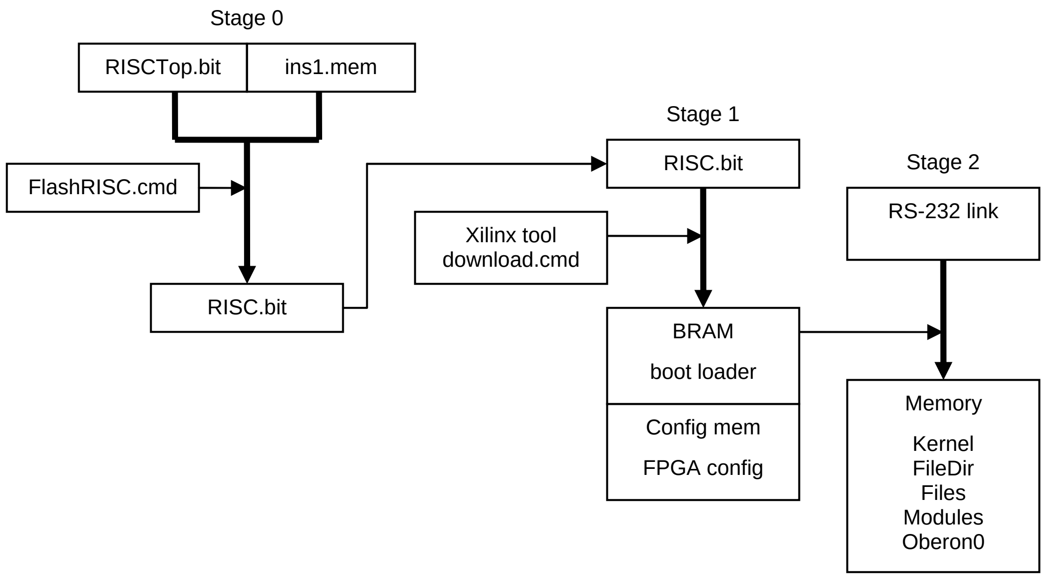


Figure 14.2 Booting from host computer

A simple boot loader reading from the RS-232 line and using the stream format described above is shown here:

```
MODULE* BootLoad;
  IMPORT SYSTEM;
  CONST MT = 12; SP = 14; MemLim = 0E7F00H;
    swi = -60; led = -60; data = -56; ctrl = -52; (*device addresses*)

  PROCEDURE RecInt(VAR x: INTEGER);
    VAR z, y, i: INTEGER;
  BEGIN z := 0;  i := 4;
    REPEAT i := i-1;
      REPEAT UNTIL SYSTEM.BIT(ctrl, 0);
      SYSTEM.GET(data, y); z := ROR(z+y, 8)
    UNTIL i = 0;
    x := z
  END RecInt;

  PROCEDURE Load;
    VAR len, adr, dat: INTEGER;
  BEGIN RecInt(len);
    WHILE len > 0 DO
      RecInt(adr);
      REPEAT RecInt(dat); SYSTEM.PUT(adr, dat); adr := adr + 4; len := len - 4 UNTIL len = 0;
      RecInt(len)
    END ;
    SYSTEM.GET(4, adr); SYSTEM.LDREG(13, adr); SYSTEM.LDREG(12, 20H)
  END Load;

BEGIN SYSTEM.LDREG(SP, MemLim); SYSTEM.LDREG(MT, 20H); SYSTEM.PUT(led, 128);
END BootLoad.
```

Another detail that must not be ignored is the handling of traps. They are implemented as a single BRL instruction, jumping conditionally to the address stored in register MT, that is, to entry 0 of the module table (which is not a module address). This address is deposited by the initialization of module *System*, which contains the trap handler. However, traps may also occur during the startup process. So, a temporary trap handler must also be installed at the very start, that is, when initializing *Kernel*.

Finally, it is worth mentioning that small Oberon programs can also be loaded and executed without the Oberon core. In fact, the boot loader is just one such example. Programs of this kind must be marked by an asterisk immediately after the symbol MODULE. This causes the compiler to generate a different starting sequence Such programs are loaded, like the boot loader in Stage 0, by the Xilinx downloader. They must not import other modules.

14.2. Building Tools

Let us summarize the prerequisites for startup:

0. The FPGA configuration and bootloader must reside in the ROM (platform flash)
1. The boot file must reside on the boot area of the disk.
2. The modules of the outer core must reside in the file system.
3. The default font and *System.Tool* must be present in the file system.

These conditions are usually met. But they are not satisfied, if either a new, bare machine is present, or if the disk store is defective. In these cases, the prerequisites must be established with the aid of suitable tools. The tools needed for the case of the bare machine or the incomplete file store are called *building tools*, those required in the case of defects are called *maintenance tools*.

Building tools allow to establish the preconditions for the boot process on a bare machine. Establishing condition 0 requires a tool for downloading the hardware configuration of the FPGA resulting from circuit synthesis, and it requires a compiler for generating the boot loader. Condition 0 is established in Stage 0.

Establishing condition 1 requires a tool for composing the boot file, and one to load it into the boot area. The former is the compiler, presumably running on a host computer. The resulting files are linked by a linker (ORL) generating a "binary" file. This file is then downloaded from the host computer to the RISC running its boot loader. Here we use an extended inner core, where the main module is not *Modules*, but *Oberon0*. The reason is that Oberon0 allows to perform the subsequent stages by accepting commands over a communication channel (here the RS-232 line). Hence, for the following stages, the tool on the RISC is Oberon0, communicating with the host computer's module ORC.

Establishing condition 2 implies the building of a file directory and the loading of files. The pair *Oberon0* and ORC contains commands for initializing a file system, for loading files over the line connection, and for moving the inner core to the disk's boot area. In addition, *Oberon0* contains further commands for file system, memory and disk inspection. Note that loading (and starting) *Oberon0* automatically starts the entire Oberon system.

There remains the important question of how *Oberon0* is loaded onto a bare machine. It is done by the boot loader with switch 1 being up. The boot file contains the inner core with the top module being *Oberon0* rather than *Modules*. The procedure is the following:

1. Select the alternative boot source by setting switch0 = 1.
2. Reset and send the boot file from the host The boot file is transferred and *Oberon0* is started.

3. Read all files from the host, (which supposedly holds all files needed for the outer core).
4. Invoke the command which loads *Oberon*. This loads the outer core, sets up the display, and starts the central loop.

A more modern solution would be to select the network as alternative boot file source. We rejected this option in order to keep net access routines outside the ROM, in order to keep the startup of a computer independent of the presence of a network and foreign sources, and also in consideration of the fact that there exist machines which operate in a stand-alone mode. As it turns out, the need for the alternative boot file source arises very rarely.

The boot linker ORL, presumably running on a host computer, where the FPGA-tools are available, is almost identical to the module loader, with the exception that object code is not deposited in newly allocated blocks, but is output in the form a file. The name of the top module of the inner core is supplied as parameter.

ORL.Link Modules	generates the regular boot file
ORL.Link Oberon0	generates the build-up boot file

Oberon0 imports two modules taking care of communication with ORL on the host computer. They are the basic module RS232, and module *PCLink1* for file transfer. The latter constitutes a task, accepting commands over the line from ORL. Their interfaces are shown below:

```
DEFINITION RS232;
    PROCEDURE Send(x: BYTE);
    PROCEDURE Rec(VAR x: BYTE);
    PROCEDURE SendInt(x: INTEGER);
    PROCEDURE SendHex(x: INTEGER);
    PROCEDURE SendReal(x: REAL);
    PROCEDURE SendStr(x: ARRAY OF CHAR);
    PROCEDURE RecInt(VAR x: INTEGER);
    PROCEDURE RecReal(VAR x: REAL);
    PROCEDURE RecStr(VAR x: ARRAY OF CHAR);
    PROCEDURE Line;
    PROCEDURE End;
END RS232.

DEFINITION PCLink1;
    PROCEDURE Run*;
    PROCEDURE Stop*;
END PCLink1.
```

The command interpreter is a simple loop, accepting commands specified by an integer followed by parameters which are either integers or names. User-friendliness was not attributed any importance at this point, and it would indeed be merely luxury. We refrain from elaborating on further details and concentrate on providing a list of commands provided by *Oberon0*. This should give the reader an impression of the capabilities and limitations of this tool module for system initiation and for error searching. (*name* stands for a string, and *a, secno, m, n* stand for integers).

	parameters	action
0	s	send and mirror s
1	a, n	show (in hex) M[a], M[a+4], ... , M[a + n*4]
2	w	fill display with words w
3	secno	show disk sector
4	filename	read file
6	-	start PC-link
7	-	show allocation, nof sectors, switches, and timer
10	-	list modules
11	modname	list commands
12	prefix	list files (enumerate directory)
13	filename	delete file

20	modname	load module
21	modname	unload module
22	name	call command
50	adr, list of values	write memory
51	adr, n	clear memory (n words)
52	secno, list of values	write sector
53	secno, n	clear sector (n words)
100	-	load boot track
101	-	clear file directory

Oberon0 imports modules *Kernel, FileDir, Files, Modules, RS232, PCLink1.* This is the inner core plus facilities for communication.

14.3. Maintenance Tools

An important prerequisite for Stage 2 (and the following stages) in the boot process has not been mentioned above. Recall that the initialization of module *FileDir* constructs the disk sector reservation table in the *Kernel* from information contained on the disk. Obviously, its prerequisite is an intact, consistent file directory. A single unreadable, corrupted file directory or file header sector lets this process fail, and booting becomes impossible. To cope with this (fortunately rare) situation, a maintenance tool has been designed: module *DiskCheck*.

DiskCheck is organized similarly to *Oberon0* as a simple command interpreter, but it imports only *Kernel* and *RS232*. Hence, booting involves only Stages 1 and 2 without any access to the disk. Operating *DiskCheck* requires care and knowledge of the structure of the file system (Chapter 7). The available commands are the following:

	parameters	action
0	s	send and mirror integer (test)
1	a, n	show (in hex) M[a], M[a+4], ... , M[a + n*4]
2	secno	show disk sector
3	secno	show head sector
4	secno	show directory sector
5	-	traverse directory
6	secno	clear header sector
7	-	clear directory (root page)

The essential command is the file directory traversal (5). It lists all faulty directory sectors, showing their numbers. It also lists faulty header sectors. No changes are made to the file system.

If a faulty header is encountered, it can subsequently be cleared (6). Thereby the file is lost. It is not removed from the directory, though. But its length will be zero.

Program *DiskCheck* must be extremely robust. No data read can be assumed to be correct, no index can be assumed to lie within its declared bounds, no sector number can be assumed to be valid, and no directory or header page may be assumed to have the expected format. Guards and error diagnostics take a prominent place.

Whereas a faulty sector in a file in the worst case leads to the loss of that file, a fault in a sector carrying a directory page is quite disastrous. Not only because the files referenced from that page, but also those referenced from descendant pages become inaccessible. A fault in the root page even causes the loss of all files. The catastrophe is of such proportions, that measures should be taken even if the case is very unlikely. After all, it may happen, and it indeed has occurred.

The only way to recover files that are no longer accessible from the directory is by scanning the entire disk. In order to make a search at all possible, every file header carries a mark field that is given a fixed, constant value. It is very unlikely, but not entirely impossible, that data sectors which happen to have the same value at the location corresponding to that of the mark, may be mistaken to be headers.

The tool performing such a scan is called *Scavenger*. It is, like *DiskCheck,* a simple command interpreter with the following available commands:

	parameters	action
0	s	send and mirror integer (test)
1	n	Scan the first n sectors and collect headers
2	-	Display names of collected files
3	-	Build new directory
4	-	Transfer new directory to the disk
5	-	Clear display

During the scan, a new directory is gradually built up in primary store. Sectors marked as headers are recorded by their name and creation date. The scavenger is the reason for recording the file name in the header, although it remains unused there by the Oberon System. Recovery of the date is essential, because several files with the same name may be found. If one is found with a newer creation date, the older entry is overwritten.

Command *W* transfers the new directory to the disk. For this purpose, it is necessary to have free sectors available. These have been collected during the scan: both old directory sectors (identified by a directory mark similar to the header mark) and overwritten headers are used as free locations.

The scavenger has proven its worth on more than one occasion. Its main drawback is that it may rediscover files that had been deleted. The deletion operation by definition affects only the directory, but not the file. Therefore, the header carrying the name remains unchanged and is discovered by the scan. All in all, however, it is a small deficiency.

Reference

1. N. Wirth. Designing a System from Scratch. *Structured Programming, 1*, (1989), 10-18.

15 Tool and service modules

In this chapter, a few modules are presented that do not belong to Oberon's system core. However, they belong to the system in the sense of being basic, and of assistance in some way, either to construct application programs, to communicate with external computers, or to analyze existing programs.

15.1. Basic mathematical functions

Module *Math* contains the basic standard functions that had been postulated already in 1960 by Algol 60. They are

sqrt(x)	the square root
exp(x)	the exponential function
ln(x)	the natural logarithm
sin(x)	the sine function
cos(x)	the cosine function

They are presented here only briefly without discussing their approximation methods. However, we point out how advantage can be taken from knowledge about the internal representation of floating-point numbers.

15.1.1. Conversion betwen integers and floating-point numbers

The Oberon System adopts the standard format postulated by IEEE. Here we restrict it to the 32-bit variant. A floating-point number *x* consistes of 3 parts

s	the sign	1 bit
e	the exponent	8 bits
m	the mantissa	23 bits

Its value is defined as $x = (-1)^s \times 2^{e+127} \times (1.m)$. A number is in normalized form, if its mantissa satisfies $1.0 \leq m < 2.0$. It is assumed that numbers are always normalized, and therefore the leading 1-bit is omitted. The exception is the singular value 0, which cannot be normalized. It must therefore be treated as a special case.

It follows that integers and floating-point numbers are represented quite differently, and that conversion operations are necessary to transfer a number from one format to the other. This is the reason why the Oberon language keeps the two types INTEGER and REAL separate. Conversion must be explicitly specified by using the two predefined functions

```
n := FLOOR(x)        REAL→ INTEGER
x := FLT(n)          INTEGER → REAL
```

Note: FLOOR(x) rounds toward -inf. For example FLOOR(1.5) = 1, FLOOR(-1.5) = -2.

The RISC processor does not feature specific instructions implementing these functions. Instead, the compiler generates inline code using the FAD instruction with special options suppressing normalization. This option is specified by the *u* and *v* modifier bits of the instruction.

The FLOOR function is realized by adding 0 with an exponent of 127 + 24 and suppressing the insertion of a leading 1-bit (u = 1). This causes the mantissa of the argument to be shifted right until its exponent is equal to 151. The RISC instructions are:

```
MOV'    R1 R0  4B00H            R1 := 4B000000H
FAD'    R0 R0 R1
```

The FLT function is implemented also by adding 0 with an exponent of 151 and forced insertion of a leading 1-bit (v = 1).

```
    MOV'    R1 R0  4B00H
    FAD"    R0 R0 R1
```

There are two predefined procedures for packing and unpacking a floating-point number:

PACK(x, e)	x := x × 2^e (for x > 0)
UNPK(x, e)	assign to *x* and *e*, such that x×2^e = x0, where *x0* is the original value of *x*, *x* becomes normalized, that is $1.0 \le x < 2.0$

Assuming R0 = x and R1 = e, the instruction sequence for PACK(x, e) is

```
    LSL     R1 R1    23
    ADD     R0 R0 R1
    STR     R0 x
```

Again assuming x = R0, the instruction sequence for UNPK(x, e) is

```
    ASR     R1 R0    23
    SUB     R1 R1   127
    STR     R1 e
    LSL     R1 R1    23
    SUB     R0 R0 R1
    STR     R0 x
```

15.1.2. The square root function

We rely on the definition x = 2^e×m Using the intrinsic UNPK procedure, the components *m* and *e* are obtained from *x*. Then the square root is computed according to the formulas

$$\mathrm{sqrt}(x) = 2^{(e \text{ DIV } 2)} \times \mathrm{sqrt}(m) \qquad \text{if e is even,}$$
$$\mathrm{sqrt}(x) = 2^{(e \text{ DIV } 2 - 1)} \times \mathrm{sqrt}(2) \times \mathrm{sqrt}(m) \qquad \text{if e is odd.}$$

The advantage is that the argument of the square root now lies in the narrow interval [1.0, 2.0], and therefore is easier and faster to approximate by a continued fracton.

```
PROCEDURE sqrt(x: REAL): REAL;
  CONST c1 = 0.70710680;   (* 1/sqrt(2) *)
    c2 = 0.590162067;
    c3 = 1.4142135;  (*sqrt(2)*)
  VAR s: REAL; e: INTEGER;
BEGIN ASSERT(x >= 0.0);
  IF x > 0.0 THEN
    UNPK(x, e);
    s := c2*(x+c1);
    s := s + (x/s);
    s := 0.25*s + x/s;
    s := 0.5 * (s + x/s);
    IF ODD(e) THEN s := c3*s END ;
    PACK(s, e DIV 2)
  ELSE s := 0.0
  END ;
  RETURN s
END sqrt;
```

15.1.3. The exponential function

Since our floating-point format is based on an exponent of 2, we first use the formula

$$\exp(x) = e^x = 2^y \quad \text{with } y = x \times \log_2(e) = x / \ln(2) \qquad \log_2(e) = 1.4426951$$

and first compute *y*. We decompose *y* into its integral part n = FLOOR(y) and its fractional part y0 = y - n. Since $2^n \times 2^{y0} = 2^{n+y0}$, the result is the sum of the exponent *n* and the mantissa 2^{y0}. Again, the advantege of the decomposition is that the argument *y0* of the polynomial approximation lies in the narrow interval [1.0, 2.0].

```
PROCEDURE exp(x: REAL): REAL;
  CONST c1 = 1.4426951;  (*1/ln(2) *)
    p0 = 1.513864173E3;
    p1 = 2.020170000E1;
    p2 = 2.309432127E-2;
    q0 = 4.368088670E3;
    q1 = 2.331782320E2;
  VAR n: INTEGER; p, y, yy: REAL;
BEGIN y := c1*x;  (*1/ln(2)*)
  n := FLOOR(y + 0.5); y := y - FLT(n);
  yy := y*y;
  p := ((p2*yy + p1)*yy + p0)*y;
  p := p/((yy + q1)*yy + q0 - p) + 0.5;
  PACK(p, n+1); RETURN p
END exp;
```

15.1.4. The logarithm

Again we take advantage of the presence of an exponent in the floating-point representation and use the equations

$$\ln (a\times b) = \ln a + \ln b$$
$$\ln (2^e\times m) = \log_2(2^e\times m) \times \ln(2) = e \times \ln(2) + \ln m$$

```
PROCEDURE ln(x: REAL): REAL;
  CONST c1 = 0.70710680;   (* 1/sqrt(2) *)
      c2 =  0.69314720;  (* ln(2) *)
      p0 = -9.01746917E1;
      p1 =  9.34639006E1;
      p2 = -1.83278704E1;
      q0 = -4.50873458E1;
      q1 =  6.176106560E1;
      q2 = -2.07334879E1;
  VAR e: INTEGER; y: REAL;
BEGIN ASSERT(x > 0.0); UNPK(x, e);
  IF x < c1 THEN x := x*2.0; e := e-1 END ;
  x := (x - 1.0)/(x + 1.0);
  y := c2 * FLT(e) + x * ((p2*x + p1)*x + p0) / (((x + q2)*x + q1)*x + q0);
  RETURN y
END ln;
```

15.1.5. The sine function

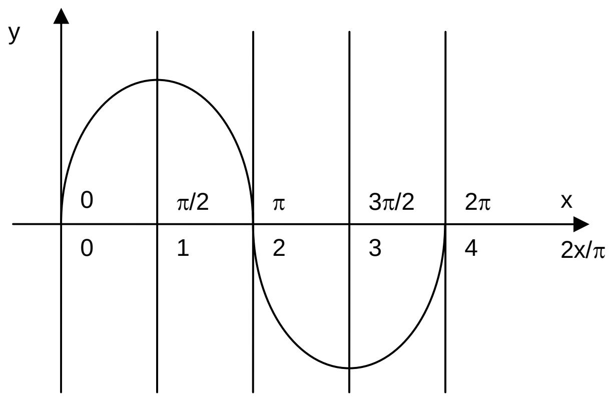


Figure 15.1 Sine function y = sin(x)

First, the argument *x* is transposed into the interval [0, π/4] by computing

```
n := FLOOR(y+0.5); y := (y - n)
```

and then distinguish between two approximating polynomials depending on whether x < π/4.

```
PROCEDURE sin(x: REAL): REAL;
  CONST c1 =  6.3661977E-1;   (*2/pi*)
    p0 =  7.8539816E-1;
    p1 = -8.0745512E-2;
    p2 =  2.4903946E-3;
    p3 = -3.6576204E-5;
    p4 =  3.1336162E-7;
    p5 = -1.7571493E-9;
    p6 =  6.8771004E-12;
    q0 =  9.9999999E-1;
    q1 = -3.0842514E-1;
    q2 =  1.5854344E-2;
    q3 = -3.2599189E-4;
    q4 =  3.5908591E-6;
    q5 = -2.4609457E-8;
    q6 =  1.1363813E-10;
  VAR n: INTEGER; y, yy, f: REAL;
BEGIN y := c1*x;
  IF y >= 0.0 THEN n := FLOOR(y + 0.5) ELSE n := FLOOR(y - 0.5) END ;
  y := (y - FLT(n)) * 2.0; yy := y*y;
  IF ODD(n) THEN f := (((((q6*yy + q5)*yy + q4)*yy + q3)*yy + q2)*yy + q1)*yy + q0
  ELSE f := ((((((p6*yy + p5)*yy + p4)*yy + p3)*yy + p2)*yy + p1)*yy + p0)*y
  END ;
  IF ODD(n DIV 2) THEN f := -f END ;
  RETURN f
END sin;
```

15.2. A data link

Module *PCLink* serves to transfer data (files) to and from another system. Data are transmitted as a sequence of blocks. Each block is a sequence of bytes. The number of data bytes lies between 0 and 255. They are preceded by a single byte indicating the length. Blocks are 255 bytes long, except rhe last block, whose length is less than 255.

Here, the transmission channel is an RS-232 line. The interface consists of two registers, one for a data byte (address = -56), and one for the status (address = -52). Bit 0 of this status register indicates, whether a byte had been received. Bit 1 of the status register indicates, whether the byte in the data register had been sent. (Note: the default transmission rate of the RISC is 9600 bit/s).

This module represents a server running as an Oberon task which must be activated by the command *Run*. A server running on the partner system must be the master issuing requests. The command sequence is a REC byte, a SND byte, or a REQ byte (for testing the connection). REC and SND must be followed by a file name, and the sequence of blocks.

Every block is acknowledged by the receiver sending an ACK byte, for which the sender waits before sending the next block. There is no synchronization within blocks. Because writing bytes onto a file may involve operations of unpredictable duration, the received bytes are not written to the file immediately. They are buffered and only output after the entire block had been received.

```
MODULE PCLink;  (*NW 8.2.2013  for Oberon on RISC*)
  IMPORT SYSTEM, Files, Texts, Oberon;
  CONST data = -56; stat = -52;
    BlkLen = 255;
    REQ = 20H; REC = 21H; SND = 22H; ACK = 10H; NAK = 11H;

  VAR T: Oberon.Task; W: Texts.Writer;
  PROCEDURE Rec(VAR x: BYTE);
  BEGIN
    REPEAT UNTIL SYSTEM.BIT(stat, 0);
    SYSTEM.GET(data, x)
  END Rec;

  PROCEDURE RecName(VAR s: ARRAY OF CHAR);
    VAR i: INTEGER; x: BYTE;
  BEGIN i := 0; Rec(x);
    WHILE x > 0 DO s[i] := CHR(x); INC(i); Rec(x) END ;
    s[i] := 0X
  END RecName;

  PROCEDURE Send(x: BYTE);
  BEGIN
    REPEAT UNTIL SYSTEM.BIT(stat, 1);
    SYSTEM.PUT(data, x)
  END Send;

  PROCEDURE Task;
    VAR len, n, i: INTEGER;
      x, ack, len1, code: BYTE;
      name: ARRAY 32 OF CHAR;
      F: Files.File; R: Files.Rider;
      buf: ARRAY 256 OF BYTE;
  BEGIN
    IF  SYSTEM.BIT(stat, 0) THEN (*byte available*)
      Rec(code);
        IF code = SND THEN  (*send file*)
          RecName(name); F := Files.Old(name);
          IF F # NIL THEN
            Send(ACK); len := Files.Length(F); Files.Set(R, F, 0);
            REPEAT
              IF len >= BlkLen THEN len1 := BlkLen ELSE len1 := len END ;
              Send(len1); n := len1; len := len - len1;
              WHILE n > 0 DO Files.ReadByte(R, x); Send(x); DEC(n) END ;
              IF ack # ACK THEN  len := 0 END
            UNTIL len1 < BlkLen
          ELSE Send(11H)
          END
        ELSIF code = REC THEN (*receive file*)
          RecName(name); F := Files.New(name);
          IF F # NIL THEN
            Files.Set(R, F, 0); Send(ACK);
            REPEAT Rec(x); len := x; i := 0;
              WHILE i < len DO Rec(x); buf[i] := x; INC(i) END ;
              i := 0;
              WHILE i < len DO Files.WriteByte(R, buf[i]); INC(i) END ;
              Send(ACK)
            UNTIL len < 255;
            Files.Register(F); Send(ACK)
          ELSE Send(NAK)
          END
        ELSIF code = REQ THEN Send(ACK) (*for testing*)
        END
    END
  END Task;

  PROCEDURE Run*;
  BEGIN Oberon.Install(T); Texts.WriteString(W, "PCLink started");
    Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END Run;

  PROCEDURE Stop*;
  BEGIN Oberon.Remove(T); Texts.WriteString(W, "PCLink stopped");
    Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END Stop;
```

```
        BEGIN Texts.OpenWriter(W); T := Oberon.NewTask(Task, 0)
        END PCLink.
```

15.3. A generator of graphic macros

The module *MacroTool* serves to create macros for the graphic system (Ch. 13). It provides the commands *OpenMacro, MakeMacro, LoadLibrary* and *StoreLibrary.*

OpenMacro decomposes the selected macro into its elements and places them at the position of the caret. This command is typically the first if an existing macro is to be modified.

MakeMacro L M collects all selected objects in the frame designated by the star pointer and unites them into macro *M*. This macro is displayed at the caret position and inserted into library *L*. If no such library exists, a new one is created.

LoadLibrary L loads the library *L* (under file name *L.Lib).* Note that a library must have been stored, before it can be loaded.

StoreLibrary stores libray *L* (with filename *L.Lib).*

The required modules are *Texts, Oberon, Graphics, GraphicFrames.*