HOST

An abstract machine for Modula-2 programs

ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

Michel Kiener, Alfred Ultsch

HOST:
AN ABSTRACT MACHINE FOR MODULA-2 PROGRAMS

February 1987

Address of the Authors:

Institut für Informatik
ETH-Zentrum
CH-8092 Zürich
Switzerland

ABSTRACT

HOST is a library of Modula-2 modules which guarantees portability of Modula-2 programs written on its top: after compilation of their sources, the same programs will run on different computers under various operating systems. However, HOST is more than yet another standard library. We call HOST an abstract machine because it offers to the programmer a coherent machine model with specific features:

- HOST incorporates a "shell" which enables users to operate programs in an easy, uniform, flexible and powerful way. HOST's shell has two seemingly contradictory functions:
  . it isolates programs from the idiosyncrasies of the specific environment
  . it integrates programs nicely into the specific environment.
- HOST enforces handy, non pompose programming style because many operations it can take care of by itself have been removed from the programmer's control and because the procedures it exports have been deliberately designed to work together well.
- HOST offers flexible error handling, either system-provided or programmer-supplied.
- HOST includes a simple yet powerful window handling capability.
- HOST is delivered with a set of utilities built on its top.

HOST has been implemented on several computers and used with success in different projects.

CONTENTS page

Acknowledgements

We are indebted to the designers of UNIX ([RITHO74], [RITHO78]), Denis M. Ritchie and Ken Thompson, and to Brian W. Kernighan and P.J. Plauger, the authors of the books "Software Tools" ([KEPL76]) and "Software Tools in Pascal" ([KEPL81]), for many ideas presented here.

Our module HEsegmentIO has been inspired from the module SIBlockIO ([BHHM86]) by E.S. Biagioni, G. Heiser, K. Hinrichs and C. Muller.

We would like to thank Hans-Jürgen Appelrath, Martin Ester, Heinrich Jasper and Helmut Lorek for many invaluable discussions and constructive criticism during HOST's design phase and for careful reading and commenting of the first draft of this paper. Thanks to Martin Odersky for valuable last minute comments.

Preliminary note

Within this paper, programmers and users all are female. This is because we love the idea of female programmers and users and they are too few within the computing community. However, we hope that HOST will seduce many male programmers and users as well.

By programmer, we mean a person writing programs which import HOST's modules. By user, we mean a user of such a program.

1. INTRODUCTION

The programming language Modula-2 has first been implemented in 1979 on a PDP-11 computer. In 1980, N. Wirth wrote a technical report containing a language definition and some basic library modules ([WIR80]). Two years later, he published "Programming in Modula-2" ([WIR82]), a book which includes among others a description of some library modules for the Lilith computer ([WIR81]). Since that time, several Modula-2 compilers appeared on the market, each of them delivered with a different set of library modules. The lack of a standard Modula-2 library (as it exists e.g. in UNIX for the C-language) has the effect that Modula-2 programs are not portable from an environment to another. A second problem of existing libraries is that they appear as an unstructured heap of modules rather than as a hierarchically ordered - and thus comprehensible - set. The common approach is "everything must be possible": it results in lots of modules exporting lots of low level procedures.

Feeling that translating calls to procedures of one library into calls to procedures of another library is an ungrateful and dispensable job and convinced that few powerful well chosen concepts are better than many pieces of junk, we decided to define a standard interface to the environment (hardware and operating system) which we called HOST, since it appears to the programmer as the only host computer she writes programs for. HOST should be viewed as an abstract machine whose machine language is Modula-2 augmented by some special purpose instructions: these are the procedures exported by the modules presented in this paper.

HOST was defined by the end of 1984 and a first version was implemented for the Lilith computer in June 1985. Since then, several software packages have been written on top of it and HOST has been ported to the SUN-3 and Macintosh computers. Still HOST remains a relatively small piece of software, so porting it from an environment to another is a straight- forward task.

In section 2 of this paper HOST's design concepts concerning program interface, module structure, programming style and window handling are presented. In section 3, we describe briefly the modules of HOST, while in

section 4, we report our experience with HOST in different projects. In section 5, we compare HOST to other Modula-2 libraries. Section 6 contains a short conclusion and a brief outlook on future work. HOST's definition modules and the definition modules of the utilities built on top of HOST and delivered with it are listed in the appendix.

2. DESIGN CONCEPTS

In this chapter, we describe the ideas that led to HOST's design concerning program interface, module structure, programming style and window handling.

2.1. Program interface

With program, we mean here a main program written on top of HOST, i.e. one that imports exclusively modules from HOST (of course, there may be any module hierachy between the main program module on top and the HOST modules at the bottom). The interface between such a program and its user is the topic of this chapter.

The crucial idea is to imbed programs into a standard environment (i.e. the "shell") which enables the user to operate them in an easy, uniform, flexible and powerful way without programming overhead: when she orders the execution of a program, she may redirect the input and output, pass arguments and options or consult the program's on line documentation. The author of the program did not need to program this features because HOST's shell put them at her disposal.

Integration into the specific environment

While users of old fashioned computers type in commands on keyboards, modern workstations' users click mouses on icons. So every computer has specific features and conventions on how it should be operated - and users dislike software that does not make usage of these features or does not respect those conventions.

For this reason, we have decided to make no prescriptions on how to implement HOST's shell on a certain computer. Each specific implementation should respect the conventions used on the underlying environment. For instance, on an IBM PC, I/O (= input/output) redirection (see below) may be initiated through typing in a redirection command, while on an Apple Macintosh, this might be achieved through a first mouse click on an icon representing a standard I/O channel and a second click on an icon representing the chosen file. As another example, passing of arguments and options (see below) might be implemented either through typing them in or selecting them from a menu.

Cooperative programs

An important insight of Software Engineering is that programs should be decomposed into modules, each module performing a well determined, specific task, and all modules working together by means of their well specified interfaces. Now why should modularity remain inside the programs? Why shouldn't we extend modularity to the program's outside? We postulate that programs shall be regarded as modules which perform well defined, specific tasks and may be connected with other programs and work together to achieve complex tasks. This way of looking at software has a beneficial influence on programmers and on the longevity of project leaders: the programmers will tend to develop sets of small, cooperative programs that perform a single job at a time, but do it well.

Files as streams of bytes

Modules have interfaces and programs have I/O. Thus, while modules work together by means of their interfaces, programs have to cooperate using each other's I/O. So the wish for programs to be able to work together in a standard way sets requirements to their I/O: the output of every program may become the input to another one, so it must have the simplest and most general possible structure, which is a stream (= a sequence) of bytes without type or structure.

Homogeneous treatment of I/O

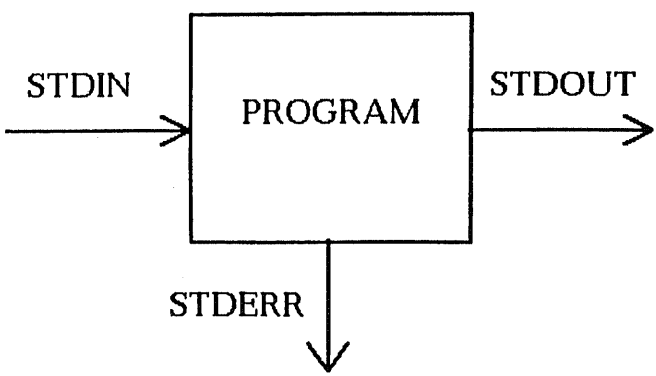
The I/O channels for programs supported by HOST are disk files, devices (terminal-keyboard and display) and windows. All of them are called "files" and treated homogeneously by HOST. While the idea of treating devices like keyboard and terminal as files stems directly from the UNIX environment, we have extended it to windows. This idea leads to a tremendous simplification of I/O handling for the programmers, since all low level I/O details are handled by HOST: programs need not care where their data comes from or goes to, and programmers need not care of the idiosyncrasies of the file system, the window manager or particular devices upon which files reside. Moreover they are delivered from the burden to remember different names of procedures that perform the same operations

on different types of files because HOST provides procedures to write to and read from files which work for all types of files, thus making I/O device independent. I/O calls that make no sense (e.g. an attempt to write to the keyboard) are ignored and an error message may be displayed. The only concepts programmers need to have in mind is that files are streams of bytes and that all types of files are treated in the same way.

From the stream of bytes nature of files, it should not be infered that files cannot have structure. Certain programs do write data in particular forms for the benefit of people or other programs. For example, a compiler creates object files in the form expected by the loader. But these structures are imposed by the programs, not by HOST.

Standard I/O and I/O redirection

A surprising number of programs have one input, one output, and perform a useful transformation on data as it passes through. This is the reason why we defined standard I/O mechanisms in HOST: standard input (STDIN) is the source from which a program will take its input. Standard output (STDOUT) is the destination for the output of a program. Standard error (STDERR) is the destination for the error messages of a program. Normally, STDIN is the terminal keyboard, STDOUT the terminal display and STDERR the terminal display too, but they can be redirected to disk files or windows.



When the user executes a program and specifies no I/O redirection, the program reads its input (if any) from the terminal keyboard (= default value of STDIN), writes its output to the terminal display (= default value of STDOUT) and, if any error occurs, writes an error message to the terminal display (= default value of STDERR). Now suppose the user wishes the same program to write its output onto a disk file instead of the terminal display. She can achieve this without the programmer having to make any change to the program: she will just have to enter a redirection command when she initiates the execution of the program.

At program execution time, the following I/O redirections are available to the user:

- redirection of input to a disk file (i.e. input is read from that file).
- redirection of output to a disk file. If that file already exists, it is overwritten.
- appending of output to a disk file (i.e. if that file already exists, the output is appended to the file after its initial content).
- redirection of error messages to a disk file (if that file already exists, it is overwritten).

The reason, why the programmer must not change a program to enable redirection of its I/O is, that programs importing HOST are embedded in a standard interface to the operating system. This interface automatically provides I/O redirection capabilities (the interface is called the "shell"). Nevertheless, it is possible to program I/O redirection as well: within a program, STDIN, STDOUT and STDERR may be redirected through simple assignement of variables of the Modula-2 type "file", e.g. "STDOUT := fl;". Whereas it is not possible for a user to redirect I/O to a window when she initiates program execution, programmers are free to program I/O redirection to windows within their programs (e.g. "STDOUT := myWindow;").

NOTE: If the user does not want to be "disturbed" by interactive error messages, she may redirect STDERR to a disk file. If the programmer does not want the user to be "disturbed" by interactive error messages, she may either program a redirection of STDERR to a disk file or provide her own error handling procedure (see section 2.5.).

Passing of arguments and options to programs

Beside the processing of redirection commands, the most important function of HOST's shell is to pass arguments and options to programs. HOST puts procedures to fetch them at the programmer's disposal.

Arguments are strings of characters used as a whole, while options are strings of characters used individually. Each single character of an option is called an option-character. As an example, "is-hungry" might be an argument (to some sensible program) and "p2a" might be an option with the option-characters "p" (e.g. standing for "protected"), "2" (e.g. meaning that the computation has to be executed twice) and "a" (e.g. standing for "abort" if an error occurs).

The way of passing arguments and options is implementation dependent: on a certain computer, the user will have to type them in at the keyboard, on another she will have to choose them with the mouse or by any other means. The details of the syntax depend on the implementation as well: on some hardware, options may be "sequences of one or more characters preceeded by a minus sign" while they may be "sequences of one or more characters entered in the option window" on another machine.

A standard argument and options passing mechanism has two main advantages:

- advantage for the users: all programs have the same standard user-interface, which is not the case when each programmer writes her own input routines
- advantage for the programmers: they are no more tempted (and are delivered from the burden) to write their own input routines.

As an example of arguments and options passing, we assume that, in a certain environment, the command

```
sqrt 17 -5n
```

activates a program sqrt, which computes the square root of the first argument (here 17) with a precision of a number of digits that is given as option (here 5 digits) and without rounding (we assume that the option-character n stands for "no rounding"). As far as STDOUT is not redirected, the result (here 4.12310) will be written to the terminal display.

On-line documentation of programs

A last feature of HOST's shell is its on-line program documentation feature. When the user initiates the execution of a program and then performs a certain, implementation dependent action (for instance clicking the mouse on the "doc" icon or entering a question mark into a commandline), HOST's shell displays the content of the program's documentation file (i.e. a documentation text) on the terminal and is ready to execute the program.

In order to be imbeded in HOST's shell, a program must contain as its first statement a call to a certain procedure which expects as a parameter the name of the program's documentation file. The advantage of this feature is to motivate programmers strongly to write documentation for their programs (this documentation should at least contain explanations about arguments and options the program expects). At the latest when a programmer first runs her program (probably to test it), the existence of the on-line documentation facility leaps to her eye. If she has not yet written a sensible text into the file the program considers to be its documentation file, she will find almost natural to provide that text right now.

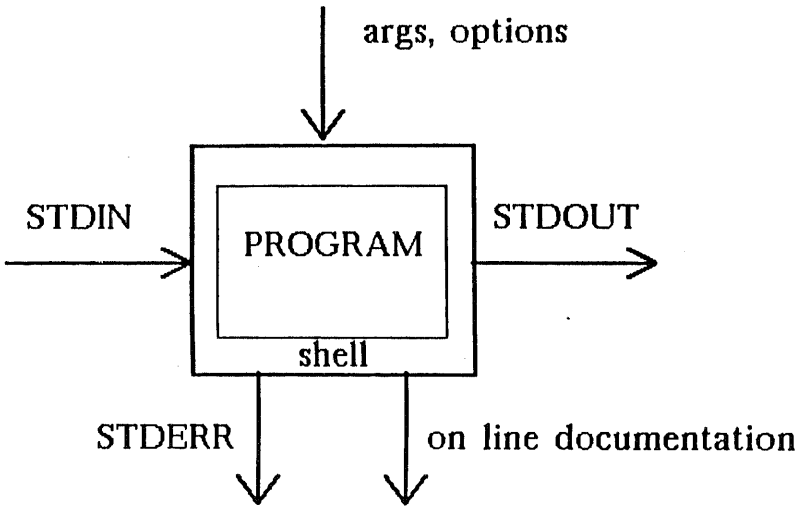
Program connection and pipes

The output of a program normally goes to STDOUT but may be redirected to a file. This file may be used as an input to another program. This is an easy method to connect programs, used for instance by multiple pass compilers.

In an environment which supports pipes (e.g. UNIX), Modula-2 programs written on top of HOST may be piped, i.e. the output of one program can be connected directly to the input of another one without having to be written onto an intermediary file.

Program model

HOST presents the following program model to programmers and users:



NOTE: Although there are only one STDIN and one STDOUT, it is of course possible to write programs with more than one input or one output on top of HOST.

2.2. The hierarchical module structure of HOST

To facilitate clear handling, we have defined four groups of Modules. Modules that belong conceptually together are in the same group and their names begin with the two same letters (we have chosen this solution because Modula-2 offers no construct to structure groups of modules without costs). The four groups of modules are:

- group HP... = HOST Primitives: Shell, files and heap management
- group HS... = HOST Strings: String manipulation
- group HD... = HOST Display: Window management
- group HE... = HOST Etc: Other modules.

Additionally, there is a module HConstypes which belongs to no group and defines implementation dependent constants and some types that are used by several modules of HOST.

Of course, programs need only to import the HOST modules they need, and do not have to pay the price (in terms of memory space) for the entire HOST.

HOST's module structure has proved to be valuable: it has remained stable since the first design.

The different modules of HOST are briefly described in section 3 and listed in the appendix.

2.3. Programming style

The style of the programs written on top of a library depends heavily on the quality of the objects exported by the library. It was a goal of HOST's design to make programmers' life as easy as possible and to enable them to write handy, non pompose programs. In this section, we first expose the different concepts used to reach this goal and then demonstrate them in a short example program.

Software Tools

All procedures exported by HOST have been designed with the "Software Tool" idea in mind: (we cite Kernighan and Plauger ([KEPL81])) "Tools solve general problems rather than special cases" (...) "Tools are designed to work together: their cumulative effect is much greater than you could get from a similar collection of programs that you could not easlily connect". Laudable is, by the way, Kernighan and Plauger's intention, "not to claim that their choices will satisfy all the programmer's needs, but should provide her with ideas and insights about her particuliar problem".

The tool concept implies uniform representation of data: all procedures that deal with strings use the same conventions about strings and all procedures that deal with real numbers use the same representation of real numbers.

Another aspect of the tool idea is handiness. HOST is a software package with manageable size and thus remains comprehensible to the programmers. It provides relatively few but well designed basic functions.

Taking some of the programmers' load off

One of our most important design decisions has been, to remove from the programmer's control the operations that our abstract machine can take care of by itself. For instance, there are no instructions to move the cursor, because we have postulated that our machine offers a mechanism (e.g. a pointing device, however it could be a set of cursor keys as well) to do that and because we think that this operation never needs to be performed by

an application program. While low level fans probably will feel frustrated that some operations escape their control, experienced application programmers will appreciate the compactness and readibility of programs which do not need to care about unnecessary details. (In the same way, some programmers consider the impossibility of incrementing processor registers a threat of Modula-2, while others see that as a chance because it enforces abstraction).

Another example is the extensive limitation testing: HOST's procedures test that the passed parameters are within the allowed range (programmers are delivered from that burden) and that the actions they are supposed to perform (e.g. closing a disk file or converting a string to a number) are performed successfully. Many procedures return a status information indicating success or failure.

A step in direction of functional programming

A lot of function procedures of HOST return one or more VAR parameters in addition to their own returned value. This is unconventional Modula-2 usage, but perfectly legal. Example:

```
PROCEDURE getcf(VAR c : CHAR; VAR f : file) : INTEGER;
```

reads a character c from a file f. It returns the read character as VAR parameter c and its INTEGER value as returned value of the function procedure. A program to read characters from a file looks like:

```
WHILE getcf(c, f) # ENDFILE DO
  (* do something sensible with c here *)
END (* WHILE *);
```

This is handy and contributes a great deal to the readibility of the code. There is only one potential drawback of this programming style: suppose we have a

```
PROCEDURE proc(VAR out : outType) : BOOLEAN;
```

which returns TRUE if a certain operation has been performed on the variable x and FALSE if that operation failed. A possible use of that

procedure is:

```
VAR x, y : outType;
  .
  .
x:=y;
IF proc(x) THEN
  (* do something sensible with x here *)
ELSE
  (* we would like to assume that x hasn't been changed by proc here
*)
  elseproc(x);
END (* IF *);
```

Here is a potential problem with the side effect of proc (changing the value of the variable x) and the else part of the if statement: proc is called anyway (because the condition part of the if statement is always executed). If proc returns FALSE, elseproc(x) is executed. Here an error which is rather difficult to discover may occur: the programmer could assume that x has the value y when it is passed to elseproc. But this is only the case if proc leaves x unchanged when it returns FALSE!

To avoid this kind of problems, we require that all function procedures with VAR parameters leave those parameters unchanged when the value they return corresponds to the less common case (e.g. when getcf returns ENDFILE or when proc returns FALSE). Thanks to this restriction, we attain both handy code and safety. (A description of the side effects in both the common and the uncommon cases is part of the definition of the function procedures of HOST).

Unread function

Many programs read sequentially characters or bytes from a file and consume them one after the other until some criteria is met. Often they have read one element too much when the criteria is met. Programs that must handle the special case of the last read element tend to be unnecessarily complicated. This is the reason why HOST provides an "unget" procedure which (logically) puts back one character or byte to the

file. This element is read again by the "getcf" or "getbf" procedure at next call (in reality, the character or byte is not written back to the file but put into a buffer). The "unget" procedure is especially practical to scan input and to write parsers.

A small example

In order to demonstrate some of the concepts praised in this section, we present the following short program part:

```
LineNumber:=1; NrOfDigits:=1;
FOR i:=0 TO 4 DO NumberString[i]:=BLANK END (* FOR *);
WHILE getline(InString, InFile, MAXSTR)>0 DO
  IF (cardtoc(LineNumber, NumberString, 5-NrOfDigits) # FAILED)
  AND appendc(BLANK, NumberString)
  AND vconcat(NumberString, InString, Outstring) THEN
      putstr(Outstring, OutFile)
  END (* IF *);
  INC(LineNumber);
  IF (LineNumber MOD 10) = 0 THEN INC(NrOfDigits) END;
END (* WHILE *);
```

This program reads from a file line by line (getline(InString, InFile, MAXSTR)), puts an increasing line number in front of the line (vconcat(NumberString, InString, Outstring)) and writes the result onto another file(putstr(Outstring,OutFile)). The line number in string form is generated from a cardinal number (cardtoc(LineNumber, NumberString, 5 - NrOfDigits)). It consists of 5 characters, is right justified and preceeded by the necessary number of spaces (BLANK). Between the line number and the input line, a space is inserted (appendc(BLANK, NumberString)). The remaining statements contain some self explaining additional details.

2.4. Windows

We have mentionned already that HOST treats windows like files and I/O devices. This means that the same procedures for reading from/writing into a file are used to read from/write to a window, and that input or output may be redirected from/to a window (only through program statement, not interactively at program execution time - see section 2.1.). Other characteristics of HOST's window management system are:

- A pointing device (e.g. a mouse) is supported.
- Moving and resizing windows are part of the HOST machine capabilities and thus not under programmers' control. Each time a window's location or dimensions are changed (and also at opening time), a programmer supplied restore-procedure is called.
- Menus and scroll bars may be attached to windows without programming overhead.
- HOST's text windows can be implemented on computers without bitmapped display and mouse.

2.5. Error handling

Often programmers spend little time in thinking about errors. In some special cases however, they want their program to react to an error in a way they determine. For this reasons, HOST offers two different kinds of error handling: system provided (= default) and programmer supplied.

All procedures exported by HOST are written so that, whenever they detect an error, they call the error procedure, passing to it the number of the occured error. Per default, the error procedure which is called is supplied by HOST, however it may be redefined by the programmer (in terms of Modula-2, the error procedure is a procedure variable which may be reassigned).

In the following, we describe how the standard error handling mechanism works and give some examples of programmer supplied error handling.

Standard error handling

Somewhere (e.g. in a text file) there is a table containing for each error number a corresponding text (e.g. "Attempt to write to a file opened for read") and a corresponding action (either "IGNORE", "ABORT" or "CONTINUE"). Whenever called, the default error procedure reads from that table the text and action entries corresponding to the error number that was passed to it. If the action is "IGNORE", the procedure does nothing, otherwise it displays the corresponding text and either aborts (when action = "ABORT") or continues (when action = "CONTINUE") the program.

Examples of programmer supplied error handling

To treat all errors in the standard way, except error number 184, we may write the following procedure:

```
PROCEDURE MyErrorProc1(ErrorNr : INTEGER);
BEGIN (* MyErrorProc1 *)
 IF ErrorNr = 184 THEN
  (* programmer supplied error handling of error 184 *)
 ELSE
  DefaultErrorProc(ErrorNr);
 END (* IF *);
END MyErrorProc1;
```

If this error procedure is to be called only at a certain location in the program, this program part may look like:

```
AssignErrorProc(MyErrorProc1);
(*here is the location of the non standard error handling *)
HOSTproc1(...);
AssignErrorProc(DefaultErrorProc);
```

Of course, it is possible to get the number of an error and use it within a program as well. To illustrate this, let us suppose we want a HOST procedure, say HOSTproc2, to be called as often as necessary until the user interactively enters a valid actual parameter x:

```
REPEAT
  GetParamFromUser(x);
  HOSTproc2(x);
UNTIL ... (* x valid *)
```

In HOST, the following are true statements:

- HOST procedures treat calls with non-valid parameters as errors, so they call the (default or programmer-supplied) error procedure
- When no error occured, no error procedure is called
- When there is an error, its number is different from NOERROR (error number "NOERROR" is reserved for "successful completion").

So our program may look like:

```
VAR ErNr : INTEGER;
      .
      .
PROCEDURE MyErrorProc2(ErrorNr : INTEGER);
BEGIN
 ErNr := ErrorNr; (* get HOST's ErNr *)
END MyErrorProc2;
      .
      .
BEGIN (* ... *)
      .
      .
 AssignErrorProc(MyErrorProc2); (* ErNr can be changed by HOST *)
 REPEAT
    ErNr := NOERROR;
    GetParamFromUser(x);
    HOSTproc2(x);
 UNTIL ErNr = NOERROR;
 AssignErrorProc(DefaultErrorProc);
      .
      .
END ...;
```

If an error occurs during execution of HOSTproc2, MyErrorProc2 will be called and ErNr will get a value different from NOERROR.

3. DESCRIPTION OF HOST'S MODULES

In this section, we describe briefly the different modules of HOST. For more details, see the definition modules in the appendix.

3.1. HPrimitives: Shell, files and heap management

HPshell

This module implements the program interface (the shell) described in section 2.1. It exports procedures to access inline arguments and options and print error messages. Further, HPshell exports the variables STDIN (= the file, all standard read procedures take their input from), STDOUT (= the file, all standard write procedures write to), STDERR (= the file where the error and message procedures write to), Keyboard (= the terminal keyboard = the file STDIN is assigned to per default) and Terminal (= the terminal display = the files STDOUT and STDERR are assigned to per default). HPshell exports the default error procedure called by all HOST procedures and a procedure to assign a programmer supplied error procedure (see section 2.5.).

HPfiles

HOST offers 4 types of files which are all treated in the same fashion: disk files, windows, terminal display and keyboard. The module HPfiles exports procedures that work for all types of files. Other procedures, which are for files of type window only, are exported by the module HDwindows.

A file (of any one of the 4 types) is a stream (sequence) of elements. Smallest element-size is one byte, largest element-size is any arbitrary user-defined record. As far as the underlying file system is concerned, a file has no internal structure: it is a featureless, contiguous stream of bytes.

All files have one of five IOmodes which is declared when a file is opened. On illegal file access, an error message appears on STDERR (see note at the end of the paragraph "Standard I/O and I/O redirection" in section 2.1.). The last read element of an input file can be put aside (unget) such that it is read again by get at next call. File I/O is normally sequential

- put and get procedures continue where the pre- ceeding call left off. This may be changed by a call to the procedure seek, which provides an easy random access capability (on disk files only). seek allows bytewise positionning of the file pointer (= actual read or write position). The actual file pointer position may be inquired through procedure getpos.

HPmemory

This module allows program controlled heap storage management and provides procedures to allocate and deallocate the heap space. The allocation is byte-wise.

3.2. HStrings: String manipulation

HOST respects the same conventions about strings as N. Wirth in [WIR82]:

- String indexing starts at 0. Thus, in the description of the procedures, the position i in an ARRAY OF CHAR refers to the (i+1)-th character in the string. (We have chosen this type of indexing because in Modula-2 the index range of an actual array parameter is mapped onto the integers 0 to N-1 (where N is the number of elements) when the array is passed to a procedure with open array parameter (see [WIR82], Page 156)).
- The length of a string str is defined as: HIGH(str) + 1, if str does not contain an ENDSTR character. Otherwise the number of characters of str up to but not including ENDSTR.
- An empty string is an ARRAY OF CHAR which contains an ENDSTR character at its first position (i.e. str[0] = ENDSTR).
- By "the last character" of a string, we mean the one preceeding the ENDSTR character, or the last one indeed if the string contains no ENDSTR.
- All procedures which append a string or a character to a string str override the ENDSTR in str with the first character to append. If possible (i.e. if the last appended character does not occupy the last place of str), an ENDSTR is appended to str after the last appended character.

HSfunctions

This module exports procedures to;

- insert, add, delete, select and modify strings
- get information about strings and
- do the lexical ordering of letters and strings.

HSconversions

This module exports procedures for conversion of strings to CARDINAL (in any base with 2 <= base <= 36) and INTEGER and vice versa.

HSreal

This module exports procedures for conversion of strings to REAL and vice versa.

3.3. HDisplay: Window management

HDwindows

This module defines operations for the following objects:

Screen: Rectangular display area on which 0, 1 or several windows can be simultaneously displayed. At one time there is one cursor to be seen somewhere on the screen and carets (see below) mark a position in the windows.

Window: A rectangular area that shows a part of a text file. To every window belongs optionally a title (line of text) and a location bar. Window may be resized by a hidden mechanism. Each time the window's dimensions are changed (and also at opening time), a programmer supplied restore-procedure is called. The actual line/column-dimensions can be inquired any time. At most one menue is associated with a window. The same procedures for reading from / writing to a file (putcf, putlf, putstr, ...) exported by HPfiles are used for reading from / writing to a window.

Cursor: A pointing mark somewhere on the screen. The cursor is moved on the screen by some input mechanisms, that is hidden in the implementation (maybe a mouse).

Menue: A menue is used to select commands. At most one menue can be seen on a screen at a time.

Caret: A caret marks a position in each window (for example to mark a position to the left of which the next character is inserted when keyboard input is done). A caret can be positioned (or its position inquired) in coordinates of lines and bytes relative to the window where the caret is in.

HDbutton

This module defines an input device with several buttons (it may but must not be a mouse). One or more of these buttons can be pressed at any time. If one of the buttons on the button-device is pressed, the procedure "pressed" exported by HDbutton returns a bitset enclosing a constant

corresponding to that button. Inquiring the state of the buttons is real time (busy-read). The buttons can, but must not, have a special meaning in connection with windows and the cursor.

HDbar

This module defines a scroll bars for windows. A scroll bar is a retangular area at the left side of a window. Pressing a button when the cursor is inside a scroll bar results in a call to a programmer supplied procedure. Installed scroll bars can be enriched by a background colored rectangle which can be dimensioned in per-mille of the total bar length. A scroll bar (incl. rectangle) is always painted and refreshed by a hidden mechanism (after changes of the outlook of the window).

3.4. HEtc: Other modules

HEreal

This module enables conversions from REAL to INTEGER and vice versa.

HEmathlib

This module exports basic mathematical functions (arc tangent, square root, exponential function, natural logarithm, cosine and sine).

HEsegmentIO

This module provides low level block I/O to mass storage and defines operations for the following objects:

Block: Blocks are arrays of bytes of fixed length.

Segment: Integral numbers of blocks grouped into so-called segments may be stored to / retrieved from so-called BIOFiles. From the programmer's point of view, segments are contiguous pieces of memory on the BIOFiles. Each segment on a particular BIOFile is uniquely identified by a SegmId which serves as the programmer's address of the segment.

BIOFile: Files of type "BIOFile" are incompatible with the files of type "file" described in the other modules of HOST (HConstypes, HPfiles, HDwindows). When created, BIOFiles get a stamp that marks them as BIOFiles. Applying procedures exported by HEsegmentIO to files that do not have that stamp produces an error message on STDERR.

3.5. HConstypes: Constants and types

The module HConstypes exports implementation dependent constants (e.g. MAXREAL = largest real number or BYTESPERCARD = size in bytes of the standard type CARDINAL) plus the types "string", "file" and "IOmode" which are used by several modules of HOST.

4. EXPERIENCE WITH HOST

As Kernighan and Plauger state, "it's not possible to learn to program well by reading platitudes about good programming, nor it is sufficient to study small examples. (The software tools have to be) real working programs that we know from experience that one can build good programms with them" ([KEPL81]).

HOST was defined by the end of 1984 and a first version was implemented for the Lilith computer in June 1985. Since then, several large software packages have been written on top of it (see 4.1.) and HOST has been ported to the SUN-3 and Macintosh computers (see 4.2.).

4.1. Software built on top of HOST

Since its first implementation, HOST has been used intensively in different projects that we briefly describe here.

KOFIS

In Summer 1984 the project KOFIS (Knowledge Based Office Information System) was started in our research group at the ETH Zürich. KOFIS ([AEJU86]) is a prototypical knowledge based personal information retrieval system running on a personal workstation. It can be viewed as an expert system for storing and retrieving documents and domain dependent knowledge in an office environmemnt. KOFIS is a large software package including a sophisticated user interface, an inference engine and a knowledge base manager. It is entirely written in Modula-2 and Prolog (the Prolog interpreter itself is written in Modula-2). The whole KOFIS software is written on top of HOST. Some of its parts have been ported from the Lilith to the Macintosh and to the SUN-3.

ODIR

In the project ODIR (Optical Disc Information Retrieval) ([APP85], [APP86]) we develop special applications and general tools for the coupling of optical disc players and personal computers. Optical discs are primarily

used to store color picture and tv-movies. The aim of the project is to develop a comfortable information retrieval system which manages database descriptors for up to 54.000 pictures stored on a disc and displays the relevant pictures when the user makes a corresponding query. The ODIR software has first been developped on the Lilith and then ported to the Macintosh without changes.

EUREKA project PROTOS

The aim of the EUREKA project PROTOS (Prolog Tools for Building Expert Systems) is to develop an integrated set of Prolog oriented tools for building expert systems, e.g. intelligent CIM-applications. Partners in PROTOS are ETH Zürich, IBM Deutschland GmbH, Sandoz AG and Universität Dortmund. The integration of two additional partners is planned.

The project consists of the four subprojects:

- Prolog programming environment
- Prolog and Databases
- Integrity concepts for Prolog systems
- Application development and tool evaluation.

The project partners have decided to use HOST as a basis for all Modula-2 software.

TOOLS

This set of utilities for programmers to sort text lines, change patterns in texts, check words against a dictionary, produce a KWIC (keywords in context) index, concatenate files, analyse the dependencies between modules within a Modula-2 program, etc., was first developped for the Lilith and then ported to the Macintosh. These tools were the first software developped on top of HOST. They are in daily use within our research group.

4.2. Portability and implementations of HOST

HOST has been designed with portability in mind and thus renounces idiosyncrasies of special hardware or software environments. For instance, HOST's text windows can be implemented on computers without bitmap-display and mouse. However, this does not mean that HOST offers only the features which all the environments have in common - which would be equivalent to offering the features of the worse one. Indeed, HOST provides the comfort, users of modern workstations with bitmap display and mouse are accustomed to. This means however, that implementing some parts of HOST on an old fashioned machine will require some extra work.

As HOST remains a relatively small piece of software (3500 lines of source code for the implementation modules, 12 kByte object code and 2 kByte data for the Lilith version), porting it from an environment to another is a straightforward task. Depending on how the implementor is familiar with HOST and with the target computer and on the quality of the available Modula-2 library on that machine, porting may take two weeks to two months.

Currently, implementations of HOST are available for the following hardware:

- Lilith
- Macintosh (4 pass compiler MacLogimo)
- Macintosh (1 pass compiler MacMETH)
- SUN 3 (SUN Modula-2)

An implementation for the IBM RT workstation is planned.

5. COMPARISON WITH OTHER MODULA-2 LIBRARIES

In this section, we present a brief overview of the Modula-2 libraries we looked at, followed by some comments about HOST's uniqueness.

The library of modules described in the Lilith Handbook ([LILI85]) was designed specially for the Lilith. Although it is compiete and includes powerful modules for screen software, it does not pretend to be a portable standard library since many modules are specific to Lilith's operating system Medos-2. We consider it more suitable for system programming than for application programming (for instance the type File exported by the module FileSystem).

Many modules of the library delivered by Logitech with their Modula-2 system for IBM PC and compatibles ([LOGI84]) are derived from the modules created for the Lilith library. A mouse, but no screen software, is supported. As in MS-DOS, there are two different types of files: binary and text.

The library of the Modula-2 for VAX/Unix BSD 4.2 system developped at Cambridge University Computer Laboratory ([CAMBR]) is organised in four sections:

1. Portable general purpose modules
2. Non-portable general purpose modules
3. Cambridge input/output
4. Unix specific modules

Only a part of the library is portable. There are lots of modules which export lots of procedures; everything seems to be possible, but it seems difficult to find what one needs. There is no screen software.

The Modula-2 library developped at OCE - Wissenschaftliches Forschungsinstitut ([OWF85]) is small, easy to use and well structured. It consists of the following modules: Conversion (of CARDINAL, INTEGER, and REAL numbers to strings and vice versa), FileIO (read and write numbers on files), Files (simple sequential and random access to

files), MathLib (set of standard mathematical functions), Storage (dynamic storage allocation and deallocation), Strings (basic string handling), TerminalIO (reading and writing characters, strings, and numbers on the standard input/output) and UnixParam (access to the parameters of a Unix program). Windows are not supported. The library is machine independent and thus portable (except UnixParam).

The Modula-2 Standard Library Definition Modules proposed by Modus, the Modula-2 Users Association ([MODU85]), were developped to be portable from a computer to another. They "provide I/O services, format conversion, mathematical functions, dynamic storage allocation and program calling. The I/O service modules tend to form layers, with some modules using services or data defined in other modules; the non-I/O modules tend to be more independent." There is no screen software.

OSSI ([BHHM86]) is a portable Modula-2 library which provides, beside the usual services, screen handling capabilities including text and graphic windows and an event manager. System dependent constants are grouped together into a module named SISystem. HOST's module HEsegmentIO has been inspired by OSSI's module SIBlockIO.

All reviewed libraries provide functions for I/O, string manipulation, number conversions, dynamic storage allocation and mathematical functions. Not all of them are portable, only the Lilith library and OSSI offer window handling procedures. Only HOST treats windows as files and claims to be portable to non-bitmap/mouse environments. Three unique features confer HOST its strong personality: its shell, its error handling mechanism and its enforcement of good programming style.

6. CONCLUSION

In our design of HOST, we have chosen the pragmatic approach: we profited from the experience of the UNIX designers ([RITHO74], [RITHO78]) and the authors of "Software Tools" ([KEPL76], [KEPL81]), because their ideas had proved to be worthwhile for real applications; further, we kept HOST small to ensure easy portability for implementors and ease of use for programmers; and, thinking that programmers have enough to do, we removed from their control the features that HOST can take care of by itself and offered them handy and powerful tools designed to work well together. Our experience with HOST in different projects has proved the soundness of this approach.

In Spring 1987, we plan to design a new version of the window handling part of HOST (i.e. the modules HDwindows, HDbar and HDbutton) that will contain an event manager enabling programmers to write programs with a so-called inverted structure: classical interactive programs control all user interaction. Often such programs consist of a main loop waiting for user input and performing some actions in response to that input. In programs with inverted structure, the control is at the event manager, which waits for events to occur and performs certain procedures (called event handlers) in response to events. Typical programs with inverted structure do not consist of straight-line code, rather they consist of a set of procedures, the event handlers. These procedures are called at the appropriate time by the event manager.

With our version of HOST's window software, programmers will have the freedom to write classical programs, programs with inverted structure and an hybrid form of both types as well, because it will be possible to pass the control from program to event manager and vice versa.

Further ameliorations of HOST's window software will include additional window, scroll-bar and menu handling capabilities and a module for graphics.

References

[APP85] H.-J. Appelrath, ODIR: Optical Disc Information Retrieval, in: Tagungsband Datenbank-Systeme für Büro, Technik und Wissenschaft (GI-Fachtagung, Karlsruhe, März 1985), Informatik-Fachberichte, Nr. 94, Springer-Verlag, Heidelberg, März 1985.

[APP86] H.-J. Appelrath, Retrieval strategies for optical disc documents, in: Proceedings of the 43rd Congress of the International Federation for Documentation, Montreal, September 1986.

[AEJU86] H.-J. Appelrath, M. Ester, H. Jasper, A. Ultsch, KOFIS: An Expert System for Information Retrieval in Offices, in: Proceedings of the "Second International Conference on the Applications of Microcomputers in Information, Documentaion and Libraries" (Baden-Baden, März 1986), North Holland Publ. Co., 1986.

[BHHM86] E. Biagioni, G. Heiser, K. Hinrichs, C. Müller, OSSI - A Portable Operating System Interface and Utility Library for Modula -2, Interner Bericht Nr. 67 des Instituts für Informatik der ETH Zürich, 1986.

[KEPL76] B. W. Kernighan and P.J. Plauger, Software Tools, Addison-Wesley, 1976.

[KEPL81] B. W. Kernighan and P.J. Plauger, Software Tools in Pascal, Addison-Wesley, 1981.

[LILI85] L. Geissmann, J. Hoppe, S. E. Knudsen, W. Winiger, B. Wagner, F. Peschel, M. Wille, W. Heiz, C. Vetterli, E. Kohen, H. Schär, J. Gutknecht, N. Wirth, Lilith Handbook, A Guide for Lilith Users and Programmers, Institut für Informatik der ETH Zürich, 1985.

[LOGI84] Logitech Modula-2/86 User's Manual, 1984, Logitech Inc., 805 Veterans Blvd., Redwood City, California 94063.

[MODU85] Modula-2 Standard Library Definition Modules, Modula-2 News, Issue # 1, January 1985, Modus (Modula-2 Users Association), PO Box 51778, Palo Alto, California 94303.

[CAMBR] Modula-2 for VAX/Unix BSD 4.2, Cambridge University Computer Laboratory, Corn Exchange Street, Cambridge, England, CB2 3QG.

[OWF85] Modula-2 Library Modules, OWF, OCE-Wissenschaftliches Forschungsinstitut AG, Zürich.

[RITHO74] D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, Communications of the ACM, July 1974, Vol. 17, Nr. 7, pp 365-375.

[RITHO78] D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, in: The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978.

[WIR80] Wirth, N., Modula-2, interner Bericht Nr. 36 des Instituts für Informatik der ETH Zürich, 1980.

[WIR81] Wirth, N., The personal computer Lilith, in: Proc. 5th Intern. Conf. on Software Engineering, San Diego, March 1981, IEEE Computer Society Press, 1981.

[WIR82] Wirth, N., Programming in Modula-2, Springer-Verlag, New York, 1982.

APPENDIX A: DEFINITION MODULES OF HOST

CONTENTS:

```
DEFINITION MODULE HPmemory;
(* HPmemory ++ allocation & deallocation of heap space *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHOR:         Alfred Ultsch
   $DATE:           February 17th, 1987        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HPmemory.def
   $MODIFICATIONS:
   $DESCRIPTION:    Procedures for program controlled heap storage management.
                    Allocation is BYTE-wise.

*)

(* Alphabetical List of Procedures: *)
(* -------------------------------- *)
(* ALLOCATE     --  allocate heap space                                      *)
(* available    --  check if heap space can be allocated                     *)
(* DEALLOCATE   --  return heap space to storage management                 *)
(* freepartial  --  return parts of an ALLOCATEd area to storage management *)
(* setmode      --  determine reactions when no more space avaiable         *)


(* Import List: *)
(* ------------ *)
FROM SYSTEM IMPORT ADDRESS;


(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED
  (* Type       *)  HPmemorymode,
  (* Procedures *)  ALLOCATE, available, DEALLOCATE, freepartial, setmode;


(* Description of exported Type: *)
(* ----------------------------- *)
TYPE
  HPmemorymode = (ABORT, CONTINUE);


(* Description of exported Procedures: *)
(* ----------------------------------- *)

PROCEDURE ALLOCATE(VAR a : ADDRESS; size : CARDINAL) : ADDRESS;
(*
 FUNCTION : allocates an area of the given size
 PARAMETER: size: number of allocated bytes of memory
 RETURNS  : a: the address of the area or NIL if no allocation is possible
*)

PROCEDURE available(size : CARDINAL) : BOOLEAN;
(*
 FUNCTION : inspects if a storage area of a given size is available
 PARAMETER: the size (in bytes) of a storage area
 RETURNS :  TRUE iff an area of the given size is available
*)

PROCEDURE DEALLOCATE(VAR a : ADDRESS; size : CARDINAL);
(*
 FUNCTION : frees the area with the given size, assigns NIL to a
```

```
 PARAMETER: a : address of the area
            size : number of bytes of a storage area
*)

PROCEDURE freepartial(a : ADDRESS; originalsize, remainingsize : CARDINAL);
(*
 FUNCTION : If a was originally a POINTER TO ARRAY [0..originalsize-1],
            it becomes a POINTER TO ARRAY [0..remainingsize-1],
            with array elements (0..remainingsize-1) being unchanged.
 NOTE:      If originalsize <= remainingsize, nothing happens.
 PARAMETER: a : address of the area
            originalsize, remainingsize : sizes in bytes of the areas
*)

PROCEDURE setmode(m : HPmemorymode);
(*
 FUNCTION : if  m  = CONTINUE, ALLOCATE returns a = NIL when not enough
            free space (default)
            if  m  = ABORT, ALLOCATE aborts when not enough free space
*)

END HPmemory.
```

```
DEFINITION MODULE HSfunctions;
(* HSfunctions ++ string functions involving strings & characters *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHOR:         Alfred Ultsch, Michel Kiener
   $DATE:           February 17th, 1987        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HSfunctions.def
   $MODIFICATIONS:
   $DESCRIPTION:    Procedures to:
                      - insert, add, delete, select and modify strings
                      - get information about strings and
                      - do the lexical ordering of letters and strings.
   $NOTE:           For the definition of a string with the maximal
                    number of characters in it, import the type
                    "string" (= ARRAY [0 .. MAXSTR-1] OF CHAR)
                    from HConstypes.
   $NOTE:           Here follow some remarks about the conventions
                    that HOST respects about the use of strings.
                    No panic, these conventions are the ones used by
                    N. Wirth and his disciples!
                1)  It is a convention of Modula-2 that the index
                    range of an actual array parameter is mapped
                    onto the integers 0 to N-1 (where N is the
                    number of elements) when the array is passed to
                    a procedure with open array parameter (see N. Wirth,
                    "Programming in Modula-2", Second Edition, Page 158).
                    This is why we have chosen the conventional indexing
                    of array elements starting at 0. Thus, in the description
                    of the procedures, the position i in an ARRAY OF CHAR
                    refers to the (i+1)-th character in the string.
                2)  An empty string is an ARRAY OF CHAR which contains
                    an ENDSTR character at its first position
                    (i.e. str[0] = ENDSTR).
                3)  The length of a string str is defined as:
                    - HIGH(str) + 1 if str does not contain an ENDSTR character
                    - the number of characters of str up to but not including
                      ENDSTR otherwise.
                4)  By "the last character" of a string, we mean the
                    one preceeding the ENDSTR character, or the last one
                    indeed if the string contains no ENDSTR.
                5)  All procedures which append a string or a character
                    to a string str override the ENDSTR in str with
                    the first character to append. If possible (i.e. if
                    the last appended character does not occupy the
                    last place of str), an ENDSTR is appended to str after
                    the last appended character.

   $NOTE:           The procedures exported by this module return only a simple
                    error status.
                    Applications requiring more detailled error informations
                    may get the error number passed to the error procedure.
                    See section 2.5. of the report:
                         HOST: An Abstract Machine for Modula-2 Programs
                         by Michel Kiener and Alfred Ultsch
                         Report of the Institut fuer Informatik der ETH Zuerich
                         February 1987.

*)


(* Alphabetical List of Procedures: *)
```

```
(* ------------------------------- *)
(* addch      -- put char in string at positon j if it fits, increment j     *)
(* appendc    -- append a character to a string                              *)
(* capitalize -- convert all small letters to capital letters                *)
(* case       -- make procedures lexorder and cmp case-sensitive             *)
(* cmp        -- compare two strings for lexical (DUDEN) ordering            *)
(* concat     -- concatenate two strings                                     *)
(* deleteNEWLINE  -- delete NEWLINE character in string                      *)
(* emptystr   -- empty a string                                              *)
(* equal      -- test two strings for equality                               *)
(* esc        -- convert escape sequence in a string into a single character *)
(* getword    -- isolate next coherent word starting at position i           *)
(* index      -- find first position of a character in a string              *)
(* insert     -- insert a substring, shift characters to make room           *)
(* isalphanum -- test if a character is a letter or a digit                  *)
(* length     -- return the length of a string                               *)
(* lexorder   -- compare two characters for lexical (DUDEN) ordering         *)
(* notcase    -- make procedures lexorder and cmp case-insensitive           *)
(* notumlaut  -- make procedures lexorder and cmp umlaute-insensitive        *)
(* scopy      -- string copy: dest[j...]:= src[i...]                         *)
(* umlaut     -- make procedures lexorder and cmp umlaute-sensitive          *)
(* NOTE: The procedures  vconcat, vcmp, vequal, vinsert and vscopy
        are the same as  concat,  cmp,  equal,  insert and  scopy
        respectively, excepted that their first parameter is a
        VAR parameter instead of a CONST parameter *)
(* vconcat    -- concatenate two strings                                     *)
(* vcmp       -- compare two strings for lexical (DUDEN) ordering            *)
(* vequal     -- test two strings for equality                               *)
(* vinsert    -- insert a substring, shift characters to make room           *)
(* vscopy     -- string copy: dest[j...]:= src[i...]                         *)


(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED

 (* - insert, add, delete, select, modify     *)
           addch,  appendc, capitalize, concat, vconcat,
           deleteNEWLINE,   emptystr,   esc,    getword,
           insert, vinsert, scopy,      vscopy,

  * - information about strings               *)
           index, isalphanum, length,

 (* - lexical ordering of letters and strings *)
           case,  notcase, cmp,      vcmp,
           equal, vequal,  lexorder, umlaut,  notumlaut;



(* Description of exported Procedures: *)
(* ----------------------------------- *)


PROCEDURE addch(ch :          CHAR;
                VAR str :     ARRAY OF CHAR;
                VAR j :       INTEGER;
                maxind :      INTEGER)          : BOOLEAN;
(*
 FUNCTION : write character ch into str[j], increment j. If j points
           outside str, j and str remain unchanged and addch returns FALSE.
```

```
 PARAMETER: maxind: the maximal index for which a writing is done
            j: is incremented when addch returns TRUE
               and remains unchanged when addch returns FALSE
 RETURNS  : TRUE iff j <= maxind and j < length(str)
 EXAMPLE  : s: ARRAY [0..20] OF CHAR;
            j:=0; WHILE addch("a", s, j, 9) DO END;
            --> s[0..9] = "aaaaaaaaaa"
                s[10..20] = (unchanged)
*)

PROCEDURE appendc(c : CHAR; VAR str : ARRAY OF CHAR) : BOOLEAN;
(*
 FUNCTION : append a character at the end of a string if possible
 PARAMETER: c: the character to append
            str: the string to which c is to append. str remains
                 unchanged if c could not be appended.
 RETURNS  : TRUE iff c could be appended
            FALSE iff length(str) >= MAXSTR or length(str) = HIGH(s) + 1
 EXAMPLE  : s: ARRAY [0..20] OF CHAR;
            s[0..2] = "abc"
            s[3] = ENDSTR
            x:=appendc("d", s);
            --> s[0..3] = "abcd"
                s[4] = ENDSTR
*)


PROCEDURE capitalize(VAR str : ARRAY OF CHAR);
(*
 FUNCTION : all letters from 'a' to 'z' are transformed into the
            corresponding capital letters; capital letters and
            non-letter characters remain unchanged
 PARAMETER: str: the string to capitalize
 EXAMPLE  : s: ARRAY [0..9] OF CHAR;
            s[0..9] = "aA1 bc%4$B"
            capitalize(s);
            --> s[0..9] = "AA1 BC%4$B"
*)


PROCEDURE case();
(*
 FUNCTION : indicates to the procedures equal, lexorder and cmp that lower-case
            letters are to be considered the same as upper-case letters.
 NOTE     : by default (i.e. as long as the procedure case() has not
            been called), lower-case letters are considered different
            from upper-case letters.
*)


PROCEDURE cmp(s1, s2 : ARRAY OF CHAR) : INTEGER;
(*
 FUNCTION : compare two strings for lexical (DUDEN) ordering.
 NOTE     : By default (i.e. as long as the procedure case() has not
            been called), lower-case letters are considered different
            from upper-case letters.
            By default (i.e. as long as the procedure umlaut() has not
            been called), umlaute letters are considered different
            from not umlaute letters.
 NOTE     : the procedures case(), notcase(), umlaut() and notumlaut()
            have an influence on the behaviour of the procedure cmp
```

```
 RETURNS  : -1 iff s1 < s2
             0 iff s1 = s2
             1 iff s1 > s2
 EXAMPLES : 1) cmp("michel", "mike") = -1
            2) cmp("zappa", "mother of invention") = +1
            3) cmp("aa", "AA") = 0 if case() wasn't called, = +1 otherwise
            4) cmp("124", "123") = +1
            5) cmp("a1", "11") = +1
*)


PROCEDURE vcmp(VAR s1 : ARRAY OF CHAR; s2 : ARRAY OF CHAR) : INTEGER;
(*
 FUNCTION : same as cmp but s1 is passed as VAR parameter
*)


PROCEDURE concat(     s   : ARRAY OF CHAR;
                      t   : ARRAY OF CHAR;
                  VAR out : ARRAY OF CHAR): BOOLEAN;
(*
 FUNCTION : concatenate "s" and "t" to "out".
 RETURNS  : TRUE iff concatenation was successful.
            If HIGH(out) is too small for out to receive the entire
            concatenated string, as much characters as possible are
            written into "out".
 EXAMPLES : 1) s, t: ARRAY [0..2] OF CHAR;
               out: ARRAY [0..20] OF CHAR;
               s[0..2] = "sss"
               t[0..2] = "ttt"
               x:=concat(s, t, out);
               --> x = TRUE
                   out[0..6] = "ssstttt"
                   out[6] = ENDSTR
            2) s, t, out: ARRAY [0..2] OF CHAR;
               s[0..1] = "ss"
               s[2] = ENDSTR
               t[0..2] = "ttt";
               x:=concat(s, t, out);
               --> x = FALSE
                   out[0..2] = "sst"
*)

PROCEDURE vconcat(VAR s   : ARRAY OF CHAR;
                  VAR t   : ARRAY OF CHAR;
                  VAR out : ARRAY OF CHAR): BOOLEAN;
(*
 FUNCTION : same as concat but s and t are passed as VAR parameters
*)


PROCEDURE deleteNEWLINE(VAR str : ARRAY OF CHAR) : BOOLEAN;
(*
 FUNCTION : replace the first NEWLINE character found in a string through
            an ENDSTR character.
 PARAMETER: str: the string from which a NEWLINE character is to delete.
 RETURNS  : TRUE iff a NEWLINE character was deleted
            FALSE otherwise
 EXAMPLE 1: s: ARRAY [0..9] OF CHAR;
            s[0..9] = a|b|c|NEWLINE|ENDSTR|...
```

```
            x := deleteNEWLINE(s);
            --> s[0..9] = a|b|c|ENDSTR|ENDSTR|...
                x = TRUE;
 EXAMPLE 2: s: ARRAY [0..9] OF CHAR;
            s[0..9] = a|b|c|d|ENDSTR|...
            x := deleteNEWLINE(s);
            --> s[0..9] = a|b|c|d|ENDSTR|...
                x = FALSE;
*)


PROCEDURE emptystr(VAR s : ARRAY OF CHAR);
(*
 FUNCTION : s is assigned an empty string (==> length(s) = 0)
 NOTE     : the implementation of this procedure is:
              s[0]:=ENDSTR;
*)


PROCEDURE equal(str1, str2 : ARRAY OF CHAR) : BOOLEAN;
(*
 FUNCTION : test the equality of two strings.
 NOTE     : By default (i.e. as long as the procedure case() has not
            been called), lower-case letters are considered different
            from upper-case letters.
            By default (i.e. as long as the procedure umlaut() has not
            been called), umlaute letters are considered different
            from not umlaute letters.
 RETURNS  : TRUE iff the strings are equal
 EXAMPLES : 1) equal("aa", "AA") = FALSE if case() has not been called
                                 = TRUE  if case() has been called
            2) equal("equal", "equal") = TRUE
*)


PROCEDURE vequal(VAR str1 : ARRAY OF CHAR; str2 : ARRAY OF CHAR) : BOOLEAN;
(*
 FUNCTION : same as equal but str1 is passed as VAR parameter
*)



PROCEDURE esc(VAR s : ARRAY OF CHAR; VAR i : INTEGER) : CHAR;
(*
 NOTE     : s is passed as VAR parameter for efficiency reasons only!
 FUNCTION : convert escape sequence in a string into a single character.
            The algorithm of this procedure is:
            IF      (i < 0) OR (i > HIGH(s))     THEN          RETURN ENDSTR;
            ELSIF   s[i] # "@"                   THEN          RETURN s[i];
            ELSE (* s[i] = "@" *)
              IF      i = HIGH(s)                THEN          RETURN "@";
              ELSIF   s[i+1] = ENDSTR            THEN          RETURN "@";
              ELSIF   s[i+1] = "n"               THEN INC(i); RETURN EOL;
              ELSIF   s[i+1] = "b"               THEN INC(i); RETURN BLANK;
              ELSIF   s[i+1] = "t"               THEN INC(i); RETURN TAB;
              ELSE                                    INC(i); RETURN s[i+1];
              END (* IF *);
            END (* IF *);
 EXAMPLES : 1) s: ARRAY [0..5] OF CHAR;
               s[0..5] = "ab@nb@"
               i = 2
```

```
                ch:=esc(s, 1);
                --> ch = EOL
                    i = 3
                    s = unchanged
            2) s: ARRAY [0..6] OF CHAR;
                s[0..6] = "ab@nb@"
                i = 5
                ch:=esc(s, 1);
                --> ch = "@"
                    i = 6
                    s = unchanged
            3) s: ARRAY [0..6] OF CHAR;
                s[0..6] = "ab@nb@"
                i = 6
                ch:=esc(s, 1);
                --> ch = ENDSTR
                    i = 6
                    s = unchanged
*)

PROCEDURE getword(VAR s :   ARRAY OF CHAR;
                      i:    INTEGER;
                  VAR out : ARRAY OF CHAR)    :  INTEGER;
(*
 NOTE     : s is passed as VAR parameter for efficiency reasons only!
 FUNCTION : isolate the next coherent seqence of non BLANK's or TAB's
            starting at s[i] and skipping leading BLANKs, TABs and NEWLINEs.
            if there is no word to get or if i points outside s,
            out is returned empty.
 RETURNS  : the index of the first character past the end of the word,
            FAILED iff there are no more characters in s or i points outside s
 NOTE     : The constant FAILED = -1 is exported by HConstypes
 EXAMPLES : 1) s: ARRAY [0..99] OF CHAR;
                out : ARRAY [0..9] OF CHAR;
                s[0..20] = "  this  is  a text   "
                x:=getword(s, 0, out);
                --> x = 6
                    out[0..3] = "this"
                    out[4] = ENDSTR
            2) s: ARRAY [0..99] OF CHAR;
                out : ARRAY [0..9] OF CHAR;
                s[0..20] = "  this  is  a text   "
                x:=getword(s, 3, out);
                --> x = 6
                    out[0..2] = "his"
                    out[3] = ENDSTR
            3) s, out: ARRAY [0..9] OF CHAR;
                s[0..9] = "hello     "
                x:=getword(s, 6, out);
                --> x = FAILED
                    out[0] = ENDSTR
            4) s, out: ARRAY [0..9] OF CHAR;
                s[0..9] = "hello     "
                x:=getword(s, 20, out);
                --> x = FAILED
                    out[0] = ENDSTR
*)


PROCEDURE index(VAR s: ARRAY OF CHAR; c : CHAR; VAR i : INTEGER) : BOOLEAN;
(*
```

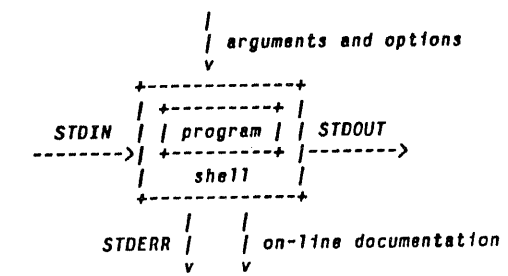
```
 NOTE      : s is passed as VAR parameter for efficiency reasons only!
 FUNCTION  : searches from left to right the string s for character c
 PARAMETER: i: the first i for which s[i] = c, FAILED if no c found in s
 RETURNS   : TRUE iff c was found in s.
 EXAMPLE   : s: ARRAY [0..4] OF CHAR;
             s[0..4] = "abcde";
             x1:=index(s, "c", i); x2:=index(s, "z", j); x3:=index(s, "a", k);
             --> i = 2
                 x1 = TRUE
                 j = FAILED
                 x2 = FALSE
                 k = FAILED
                 x3 = TRUE
 NOTE      : The constant FAILED = -1 is exported by HConstypes
*)


PROCEDURE insert(src      : ARRAY OF CHAR;
                 VAR dest : ARRAY OF CHAR;
                 j        : INTEGER)        : BOOLEAN;
(*
 FUNCTION : insert the substring str into the string dest, starting at
            position dest[j] and after having shifted the original characters
            dest[j], dest[j+1], dest[j+2], ..., dest[j + length(src) - 1]
            to the right to make room. src is copied from its first character
            up to its last one. If dest is too short, only as much characters
            as possible are inserted and shifted.
            If j points outside dest, dest remains unchanged and
            insert returns FALSE.
 RETURNS   : TRUE iff all characters could be inserted and shifted
             FALSE iff HIGH(dest) + 1 < length(src) + j
 EXAMPLES : 1) dest : ARRAY [0..9] OF CHAR;
               dest[0..6] = "abcfgh"
               dest[6] = ENDSTR
               x:=insert("de", dest, 3);
               --> x = TRUE
                   dest[0..7] = "abcdefgh"
                   dest[8] = ENDSTR
            2) dest : ARRAY [0..9] OF CHAR;
               dest[0..6] = "abcfgh"
               dest[6] = ENDSTR
               x:=insert("12345", dest, 3);
               --> x = FALSE
                   dest[0..9] = "abc12345fg"
            3) dest : ARRAY [0..9] OF CHAR;
               dest[0..6] = "abcfgh"
               dest[6] = ENDSTR
               x:=insert("12345678", dest, 3);
               --> x = FALSE
                   dest[0..9] = "abc1234567"
*)

PROCEDURE vinsert(VAR src  : ARRAY OF CHAR;
                  VAR dest : ARRAY OF CHAR;
                  j        : INTEGER)        : BOOLEAN;
(*
 FUNCTION : same as insert but src is passed as VAR parameter
*)


PROCEDURE isalphanum(c : CHAR) : BOOLEAN;
```

```
(*
 RETURNS : TRUE iff c in [a..z, A..Z, 0..9, ä, ö, ü, Ä, Ö, Ü]
*)


PROCEDURE length(VAR s :  ARRAY OF CHAR) : CARDINAL;
(*
 NOTE     : s is passed as VAR parameter for efficiency reasons only!
 RETURNS  : the length of the string s, which is:
              HIGH(s) + 1 iff s does not contain an ENDSTR character
              the number of characters up to but not including ENDSTR otherwise
 EXAMPLES : 1) s : ARRAY [0..4] OF CHAR;
               s[0..4] = "abcde"
               --> length(s) = 5
            2) s : ARRAY [0..4] OF CHAR;
               s[0..1] = "AB"
               s[2] = ENDSTR
               s[3..4] = "CD"
               --> length(s) = 2
*)


PROCEDURE lexorder(c1, c2 : CHAR) : INTEGER;
(*
 FUNCTION : compare two characters for lexical (DUDEN) ordering
 NOTE     : By default (i.e. as long as the procedure case() has not
            been called), lower-case letters are considered different
            from upper-case letters.
            By default (i.e. as long as the procedure umlaut() has not
            been called), umlaute letters are considered different
            from not umlaute letters.
 NOTE     : the procedures case(), notcase(), umlaut() and notumlaut()
            have an influence on the behaviour of the procedure lexorder
 RETURNS  : -1 iff c1 < c2
             0 iff c1 = c2
             1 iff c1 > c2
*)


PROCEDURE notcase();
(*
 FUNCTION : indicates to the procedures equal, lexorder and cmp that lower-case
            letters are to be considered different from upper-case letters.
 NOTE     : by default (i.e. as long as the procedure case() has not
            been called), lower-case letters are considered different
            from upper-case letters.


PROCEDURE notumlaut();
(*
 FUNCTION : indicates to the procedures equal, lexorder and cmp that umlaute
            letters (i.e. ä, ö and ü) are to be considered the same
            as the corresponding not umlaute letters (i.e. a, o and u).
 NOTE     : by default (i.e. as long as the procedure umlaut() has not
            been called), umlaute letters are considered different
            from not umlaute letters.
*)
```

```
PROCEDURE scopy(src      : ARRAY OF CHAR;
                i        : INTEGER;
                VAR dest : ARRAY OF CHAR;
                j        : INTEGER)        : BOOLEAN;
(*
 SHORT DESCRIPTION: string copy: dest[j...]:= src[1...]
 FUNCTION : copy src from its (i + 1)-th character up to its last one
            into dest, with the first copied character put into dest[j].
            If dest is too short, only as much characters as possible
            are copied.
            If i points outside src or j points outside dest, dest
            remains unchanged and scopy returns FALSE.
 RETURNS  : TRUE iff all characters could be copied
            FALSE iff HIGH(dest) + 1 - j < length(src) - i
 EXAMPLES : 1) src : ARRAY [0..4] OF CHAR;
               dest : ARRAY [0..4] OF CHAR;
               src[0..4] = "abcde"
               x:=scopy(src, 1, dest, 0);
               --> x = TRUE
                   dest[0..3] = "bcde"
                   dest[4] = ENDSTR
            2) src : ARRAY [0..9] OF CHAR;
               dest : ARRAY [0..4] OF CHAR;
               src[0..6] = "abcdefg"
               src[7] = ENDSTR
               x:=scopy(src, 1, dest, 3);
               --> x = FALSE
                   dest[0..2] = (unchanged)
                   dest[3..4] = "bc"
            3) src, dest : ARRAY [0..9] OF CHAR;
               src[0..6] = "abcdefg"
               src[7] = ENDSTR
               x:=scopy(src, 1, dest, 0);
               --> x = TRUE
                   dest[0..5] = "bcdefg"
                   dest[6] = ENDSTR
*)


PROCEDURE vscopy(VAR src  : ARRAY OF CHAR;
                 i        : INTEGER;
                 VAR dest : ARRAY OF CHAR;
                 j        : INTEGER)        : BOOLEAN;
(*
 FUNCTION : same as scopy but src is passed as VAR parameter
*)



PROCEDURE umlaut();
(*
 FUNCTION : indicates to the procedures equal, lexorder and cmp that umlaute
            letters (i.e. Ä, Ö and Ü) are to be considered different
            from the corresponding not umlaute letters (i.e. a, o and u).
 NOTE     : by default (i.e. as long as the procedure umlaut() has not
            been called), umlaute letters are considered different
            from not umlaute letters.
*)
```

```
END HSfunctions.
```

```
DEFINITION MODULE HPshell;
(* HPshell ++ processing of STD-files, arguments & options; error handling *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHOR:         Alfred Ultsch, Michel Kiener
   $DATE:           February 17th, 1987        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HPshell.def
   $MODIFICATIONS:
   $DESCRIPTION:    This module defines a program interface (called "shell")
                    which enables the user to redirect I/O, pass arguments
                    and options and activate on-line program documentation.
                    It exports procedures to enter and leave the shell,
                    access the arguments and options passed by the user,
                    assign the terminal keyboard or the terminal display
                    to a file and display error messages onto the standard
                    error channel.
                    Further, this module exports error handling procedures,
                    including the default error procedure called by all
                    HOST procedures whenever an error occurs and a procedure
                    to assign a programmer supplied error procedure.
                    The concepts behind HPshell are described in sections
                    2.1. and 2.5. of the report:
                      HOST: An Abstract Machine for Modula-2 Programs
                      by Michel Kiener and Alfred Ultsch
                      Report of the Institut fuer Informatik der ETH Zuerich
                      February 1987.

                    A main program which has called EnterShell is connected to
                    the outer world according to the following picture:
```




```
                    and has the following properties:
                    1) It is connected to standard I/O channels: standard
                       input (STDIN) is the source from which the program
                       takes its input. Standard output (STDOUT) is the
                       destination for the output of the program.
                       Standard error (STDERR) is the destination
                       for the error messages of the program. Per default
                       STDIN is the terminal keyboard, STDOUT the terminal
                       display and STDERR the terminal display
                       too, but they can be redirected to disk files or
                       windows.
                    2) Disk files, devices (Keyboard and Terminal) and
                       windows are all treated in the same way and are
                       all considered to be files.
                    3) It is possible to redirect I/O: STDIN, STDOUT and
                       STDERR can be redirected to devices (Terminal,
                       Keyboard) disk files or windows:
```

a) either when the user starts a program and passes a redirection command
b) or in the program through the assignments "STDIN := f;", "STDOUT := f;" and "STDERR := f;". (e.g. STDOUT := f; assigns STDOUT to file f, i.e. sends standard output to f). If f is a disk file, it is the programmer's responsibility to make sure that it has be opened through a call of HPfiles.open(...) for an appropriate IOmode (IOEEAD or IFTHERE for reading and IOWRITE, IOAPPEND or IOVERRIDE for writing).
If f has not been opened properly or if it has been opened for a wrong IOmode, an error message will appear on STDERR when the next I/O operation is performed.
It is the programmer's responsibility to close an old standard I/O when redirecting to a new one (i.e. if STDOUT = f1, and then "STDOUT := f2;" is performed, then f1 has to be closed before the program is finished).
Redirecting a standard I/O to the same file it was before (i.e. "AssignTerminal(STDOUT)" when STDOUT already was equal to Terminal) has no effect.
The standard I/Os that are actually assigned to STDIN, STDOUT and STDERR should all be closed at the end of a program through a call of HPshell.LeaveShell().

4) The user may pass arguments and options to the program at execution time. It is the programer's responsibility to have arguments and options fetched by her program through corresponding calls to the procedures getarg and getopt. If the program does not fetch some or all of the arguments or options, nothing happens (i.e. the program runs without considering the unused arguments or options). Options are a special kind of arguments composed of a sequence of so called option-characters.

$NECESSARY CONDITION: To ensure opening and closure of the shell, the very first and last statements of a main program ppp written on top of HOST must be calls to EnterShell and LeaveShell:

```
MODULE ppp;
(*...*)
BEGIN (* ppp *)
  EnterShell("ppp.MAN");
  (*...*)
  (* <ppp's program-text> *)
  (*...*)
  LeaveShell();
END ppp.
```

```
    ------------------------------------------------------------------------
    | NOTE: All procedures exported by HOST call the (default or           |
    |       programmer-supplied) error procedure ONLY WHEN AN ERROR OCCURS.|
    |       When completion is  successful, NO ERROR PROCEDURE IS CALLED   |
    |       (because this would consume computing time for nothing).       |
    |                                                                      |
    |       Programmers must be aware of the following when they use error |
    |       numbers:                                                       |
    |       - the constant NOERROR is exported for programmers'            |
    |         convenience only                                             |
    |       - when an error occurs, its error number is different from     |
    |         NOERROR                                                      |
    |       - when no error occurs, NOERROR is NOT the value passed to the |
    |         error procedure, because when no error occurs, the error     |
    |         procedure is not called at all!                              |
    |                                                                      |
    |       An example how to get the error number and use it within a     |
    |       program can be found in section 2.5. of the report:            |
    |           HOST: An Abstract Machine for Modula-2 Programs            |
    |           by Michel Kiener and Alfred Ultsch                         |
    |           Report of the Institut fuer Informatik der ETH Zuerich     |
    |           February 1987.                                             |
    ------------------------------------------------------------------------
*)


(* Alphabetical List of Procedures: *)
(* ------------------------------- *)
(* AssignErrorProc  -- assign a programmer defined error procedure      *)
(* AssignKeyboard   -- assign terminal keyboard to file                 *)
(* AssignTeminal    -- assign terminal display to file                  *)
(* DefaultErrorProc -- default proc called by HOST when an error occurs *)
(* EnterShell       -- enter the shell                                  *)
(* error            -- print a message and exit the program             *)
(* getarg           -- get an argument entered by the user              *)
(* GetErrorProc     -- return current error procedure                   *)
(* GetErrorText     -- get error text and action from error table       *)
(* getopt           -- get an option-character entered by the user      *)
(* LeaveShell       -- leave the shell                                  *)
(* message          -- print a message and continue                     *)
(* nargs            -- return the number of arguments                   *)



(* Import List: *)
(* ------------ *)
FROM HConstypes  IMPORT (* type *) file;


(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED

(* Constant: *)
   NOERROR, (* When there is an error, its number is different from NOERROR.
               SEE NOTE ABOVE!                                          *)

(* Types: *)
   ErrorAction,     (* type of the action to undertake when an error occurs *)
```

```
   ErrorProcType,  (* type of the error procedures                    *)

(* Variables: *)
   STDIN,        (* the file, all standard read procs take their input from *)
   STDOUT,       (* the file, all standard write procs write to             *)
   STDERR,       (* the file where the error and message procs write to     *)

(* Procedures: *)
   AssignKeyboard,          (* assign keyboard to file          *)
   AssignTerminal,          (* assign terminal display to file  *)
   EnterShell, LeaveShell,  (* enter and leave the shell        *)
   getarg, getopt, nargs,   (* commandline decoding             *)
   error, message,          (* hints for the user               *)
   GetErrorText,            (* get error text and action        *)
   GetErrorProc,            (* return error procedure           *)
   AssignErrorProc,         (* assign error procedure           *)
   DefaultErrorProc;        (* default error procedure          *)


(* Declaration of exported Objects: *)
(* -------------------------------- *)

CONST
   NOERROR = 0;

TYPE
   ErrorAction   = (ABORT, CONTINUE, IGNORE);
   ErrorProcType = PROCEDURE (INTEGER);

VAR
  STDIN, STDOUT, STDERR : file;


(* Declaration of exported Procedures: *)
(* ----------------------------------- *)


PROCEDURE AssignErrorProc(ErrorProc : ErrorProcType);
(*
 FUNCTION:   assign error procedure
 PARAMETER:  ErrorProc : the error procedure that will be called by the
                         procedures exported by HOST when they dectect an error
*)


  OCEDURE AssignKeyboard(VAR ToFile : file);

  UNCTION  : ToFile := keyboard;
             assign the terminal keyboard (which is the file STDIN is
             assigned to per default) to the file ToFile
 PARAMETER : ToFile : the file the terminal keyboard is assigned to
*)


PROCEDURE AssignTerminal(VAR ToFile : file);
(*
 FUNCTION  : ToFile := terminal;
             assign the terminal display (which is the file STDOUT is
             assigned to per default) to the file ToFile
 PARAMETER : ToFile : the file the terminal display is assigned to
*)
```

```
PROCEDURE DefaultErrorProc(ErrorNr : INTEGER);
(*
 FUNCTION  : default proc called by HOST when error ErrorNr occurs
 PARAMETER : ErrorNr: the number of the occured error
*)


PROCEDURE EnterShell(DocFileName: ARRAY OF CHAR);
(*
 FUNCTION : Enter the shell, proceed the arguments and options and the
            redirection commands. If the user requests on-line
            documentation, print the content of the file with name
            DocFileName on STDOUT and continue. If no such file exists,
            print an error message and continue.
 PARAMETER: DocFileName: the name of the documentation file
 NOTE :     a call to EnterShell must be the first statement of any main
            program
*)


PROCEDURE error(msg : ARRAY OF CHAR);
(*
 FUNCTION: write the message msg on STDERR, then perform HALT
*)


PROCEDURE getarg(       n : INTEGER;
                 VAR str : ARRAY OF CHAR;
                 maxsize : INTEGER) : BOOLEAN;
(*
 FUNCTION:   Get the n-th commandline argument. The first argument on the
             commandline is number one.
 PARAMETERS: n: number of the commandline argument to fetch.
             maxsize: if maxsize is smaller than the length of
                      the actual parameter str, the returned argument
                      is truncated to this length.
             str: if getarg returns TRUE, str contains the argument
                  if getarg returns FALSE, str remains unchanged.
 RETURNS:    TRUE if the argument is present, FALSE otherwise.
*)


PROCEDURE GetErrorProc(VAR CurrentErrorProc : ErrorProcType);
(*
 FUNCTION:  Return the current error procedure, e.g. to be included in a new one
 PARAMETER: CurrentErrorProc : current error procedure
*)


PROCEDURE GetErrorText(ErrorNr : INTEGER;
                       VAR Text : ARRAY OF CHAR;
                       VAR Action : ErrorAction);
(*
 FUNCTION:   read from the error table the text and the action
             corresponding to the error number ErrorNr
 PARAMETERS: ErrorNr : the number of the error
             Text    : the corresponding text (is truncated to the length
```

```
                    of the actual parameter Text if necessary)
            Action  : the corresponding text
*)


PROCEDURE getopt(argnr : INTEGER; optchar : CHAR) : INTEGER;
(*
 FUNCTION:   Look if the character optchar is present on the commandline
             as part of an argument starting with the character
             OPTION ("-").
 PARAMETERS: argnr : the argument number on the commandline, from where on
                     to search.
             optchar: the character, that is supposed to be present
 RETURNS:    The number of the argument, where optchar was found.
             FAILED if it was not found (the constant FAILED is exported
             by HConstypes and has the value -1).
*)


PROCEDURE LeaveShell();
(*
 FUNCTION : leave the shell
 NOTE :     a call to LeaveShell must be the last statement of any main
            program
*)


PROCEDURE message(msg : ARRAY OF CHAR);
(*
 FUNCTION: write the message msg to STDERR, then continue
*)


PROCEDURE nargs() : INTEGER;
(*
 FUNCTION: return the number of arguments that have been entered by the user
*)


END HPshell.
```

```
DEFINITION MODULE HPfiles;
(* HPfiles ++ file streams & get, put of BYTE, CHAR, RECORD and string *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHOR:         Alfred Ultsch, Michel Kiener
   $DATE:           February 17th, 1987        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HPfiles.def
   $MODIFICATIONS:
   $DESCRIPTION:    - HOST offers 4 types of files which are all treated
                      in the same fashion: disk files, windows, terminal
                      display and keyboard. The data type "file"
                      and some constants returned or used by the procedures
                      of HPfiles (e.g. ENDSTR, ENDFILE, etc.) are exported
                      by the module HConstypes.
                    - Each variable of type "file" has a "filename"-field.
                      When a file is opened the actual "name" is copied
                      into the "filename"-field of the file variable, so that
                      it can be asked later.
                    - A file (of any one of the 4 types) is a stream
                      (sequence) of elements. Smallest element-size is
                      one byte, largest element-size is any arbitrary
                      user-defined record. As far as the underlying file
                      system is concerned, a file has no internal structure:
                      it is a featureless, contiguous stream of bytes.
                    - All files have one of five IOmodes: IOREAD, IOWRITE,
                      IOAPPEND, IOVERRIDE, or IFTHERE, which is declared when
                      a file is open-ed. For disk files, the IOmode can be changed
                      through a call to the procedure seek. The IOmodes are:
                      IOREAD:     files are sequences of elements from which
                                  elements may be read (get) in a serial manner.
                      IOWRITE:    files are sequences of elements to which
                                  elements may be written (put) in a serial
                                  manner. If the file exists already, the
                                  user is asked if she wants to override it.
                      IOAPPEND:   the same as IOWRITE, but eventual contents of
                                  the file are not removed and new stuff is
                                  appended at the end of the file.
                      IOVERRIDE:  the same as IOWRITE, but eventual existing file
                                  is overridden without user-confirmation.
                      IFTHERE:    the same as IOREAD but no action is taken if
                                  the file cannot be opened.
                      On illegal file access (e.g. attempt to write on a file
                      opened with IOREAD) the error procedure of HPshell is
                      called.
                    - The last read byte of an input file (of a file
                      opened for IOREAD or IFTHERE) can be put aside (unget)
                      such that it is read again by getcf or getbf at next call.
                    - File I/O is normally sequential - put and get procedures
                      continue where the preceeding call left off. This
                      may be changed by a call of the procedure seek,
                      which provides an easy random access capability - on disk
                      files only -. seek allows bytewise positionning of
                      the file pointer (= actual read or write position).
                      The actual file pointer position may be asked through
                      procedure getpos.
                    - CHAR values that are read/written may be interpreted or
                      changed in values by the files devices, for example by
                      coercing CR, LF to NEWLINE. BYTE values remain unchanged.

   $NOTE:           The procedures exported by this module return - if any -
                    only a simple error status (TRUE indicating success and
                    FALSE failure).
```

```
                    Applications requiring more detailled error informations
                    may get the error number passed to the error procedure.
                    See section 2.5. of the report:
                      HOST: An Abstract Machine for Modula-2 Programs
                      by Michel Kiener and Alfred Ultsch
                      Report of the Institut fuer Informatik der ETH Zuerich
                      February 1987.

*)

(* Alphabetical List of Procedures: *)
(* ------------------------------- *)
(* busyget   -- check if a byte is currently at Keyboard                    *)
(* close     -- close an open disk file                                     *)
(* EchoOff   -- characters typed in at Keyboard are NOT echoed at Terminal  *)
(* EchoOn    -- characters typed in at Keyboard are echoed at Terminal      *)
(* getbf     -- get a BYTE         from a file                              *)
(* getcf     -- get a CHAR         from a file                              *)
(* getline   -- get a line of text from a file                              *)
(* getpos    -- get actual file pointer position of disk file               *)
(* getrf     -- get a RECORD       from a file                              *)
(* lengthf   -- return length in bytes of a disk file                       *)
(* open      -- associate a filename to the file and specify IOmode         *)
(* putbf     -- put a BYTE    into a file                                   *)
(* putcf     -- put a CHAR    into a file                                   *)
(* putlf     -- put a NEWLINE into a file                                   *)
(* putrf     -- put a RECORD  into a file                                   *)
(* putstr    -- put a string  into a file                                   *)
(* remove    -- remove a disk file                                          *)
(* rename    -- rename an open file                                         *)
(* seek      -- position bytewise in a disk file                            *)
(* ungetf -- put last read BYTE aside so that getbf or getcf will read it again *)

(* Import List: *)
(* ------------ *)
FROM HConstypes  IMPORT (* typ   *) BYTE, IOmode, file;



(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED
 (* Procedures: *)
 (* - installation of files  *) close,   open,    remove,  rename,  lengthf,
 (* - busyread from Keyboard *) busyget,
 (* - input                  *) getbf,   getcf,   getrf,   getline,
 (* - output                 *) putbf,   putcf,   putrf,   putstr,  putlf,
 (* - put back one byte      *) ungetf,
 (* - positioning            *) seek,    getpos,
 (* - echo on/off            *) EchoOff, EchoOn;


(* Declaration of exported Procedures: *)
(* ----------------------------------- *)

PROCEDURE busyget(VAR b : BYTE) : INTEGER;
(*
 FUNCTION : get the next character from the terminal keyboard,
            but return immediately if no character has been typed.
 PARAMETER: b: 0C if no character has been typed, the read character otherwise.
 RETURNS  : the integer value of b or FAILED if no character has been typed.
```

```
*)

PROCEDURE close(VAR f : file) : BOOLEAN;
(*
 FUNCTION : close the file f
 RETURNS  : TRUE if file was closed, FALSE otherwise
 NOTE     : the destiny of unclosed disk files is environment dependent!
*)

PROCEDURE EchoOff();
(*
 FUNCTION : After this procedure has been called, characters typed in at
            the terminal keyboard are not echoed to the terminal display.
*)

PROCEDURE EchoOn();
(*
 FUNCTION : After this procedure has been called, characters typed in at
            the terminal keyboard are echoed to the terminal display.
*)


PROCEDURE getbf(VAR b : BYTE; VAR f : file) : INTEGER;
(*
 FUNCTION : read a byte from a file
 PARAMETER: b: the read byte, 0C if there is no (more) byte to read.
            f: the file from which to read.
 RETURNS  : the integer value of the read byte if possible or
            ENDFILE if there is no more byte to read.
*)


PROCEDURE getcf(VAR c : CHAR; VAR f : file) : INTEGER;
(*
 FUNCTION : read a character from a file
 PARAMETER: c: the read character, 0C if there is no (more) character to read.
            f: the file from which to read.
 RETURNS  : the integer value of the read character if possible or
            ENDFILE if there is no more character to read.
*)

PROCEDURE getline(VAR line : ARRAY OF CHAR;
                  VAR infile : file;
                  maxlength : INTEGER) : INTEGER;
(*
 FUNCTION:  read sequentially characters from the file "infile"
            and put them into line starting at line[0] and until
            1) either NEWLINE or end-of-file is encountered
            2) or maxlength characters have been read, or line is already full.
               In this case (i.e. if  maxlength characters have been
               read, or line is already full) read and skip the next characters
               until NEWLINE or ENDFILE is encountered (so that the next
               call of getline will really get the next line!).
            Then append an ENDSTR to line if possible.
 NOTE:      The returned line does contain a NEWLINE character if one
            was read and the size of the actual line parameter and
            maxlength are big enough.
 NOTE:      If the length of the actual parameter line is not sufficient,
            the returned line will be truncated.
 NOTE:      To eliminate the NEWLINE character in a string, use the
```

```
            procedure HSfunctions.deleteNEWLINE(...).
 NOTE:      The escape character is treated by getline like all the other
            characters. If special treatment of the escape character is
            wanted, it has to be programmed using procedure getcf.
 PARAMETER: line:      place, where the resulting line is stored to. If
                       end-of-file was encountered, line is returned empty
                       (i.e. line[0] = ENDSTR).
            infile:    the file from which it is read
            maxlength: the returned line is truncated to this length if
                       necessary
 RETURNS :  the number of read characters (NEWLINE included), = 0 implies
            end-of-file
 EXAMPLE 1) Suppose the file infile contains:
               a|b|c|NEWLINE|d|e|NEWLINE|f|g|ENDFILE
               ↑                 ↑
               byte number 0     byte number 5
            and we have:
               VAR line: ARRAY [0..20] OF CHAR; infile: file; x: INTEGER;
               x:= getline(line, infile, 30);
               --> x = 4, line[0..2] = "abc", line[3] = NEWLINE;
                   line[4] = ENDSTR.
               x:= getline(line, infile, 30);
               --> x = 3, line[0..1] = "de", line[2] = NEWLINE;
                   line[3] = ENDSTR.
               x:= getline(line, infile, 30);
               --> x = 2, line[0..1] = "fg", line[2] = ENDSTR.
               x:= getline(line, infile, 30);
               --> x = 0, line[0] = ENDSTR.
 EXAMPLE 2) Suppose the file infile contains:
               a|b|c|NEWLINE|d|e|NEWLINE|f|g|ENDFILE
            and we have:
               VAR line: ARRAY [0..20] OF CHAR; infile: file; x: INTEGER;
               x:= getline(line, infile, 2);
               --> x = 2, line[0..1] = "ab", line[2] = ENDSTR.
               x:= getline(line, infile, 3);
               --> x = 3, line[0..1] = "de", line[2] = NEWLINE;
                   line[3] = ENDSTR.
*)

PROCEDURE getpos(VAR highpos : CARDINAL; VAR lowpos : CARDINAL; VAR f : file);
(*
 FUNCTION : get actual file pointer position of disk file f.
            The actual file pointer position is the byte number
            (highpos*MAXCARD+lowpos). Byte numbers
            are counted from the beginning of the file starting with zero.
 NOTE     : on the SUN, highpos is always returned with a value = 0.
*)

PROCEDURE getrf(VAR record : ARRAY OF BYTE; VAR f : file) : INTEGER;
(*
 FUNCTION : read a record from file f. The number of BYTEs read is
            determined by the size of the actual parameter "record".
 RETURNS  : the number of BYTEs read
            or 0 if the read was not successful
            or ENDFILE, if there is no more record to read.
*)


PROCEDURE lengthf(VAR highpos, lowpos: CARDINAL; VAR f : file);
(*
 FUNCTION : the length of file f in bytes is (highpos * MAXCARD + lowpos)
```

```
 NOTE      : on the SUN implementation, the returned highpos is always  = 0.
*)

PROCEDURE open(name : ARRAY OF CHAR; mode : IOmode; VAR f : file) : BOOLEAN;
(*
 FUNCTION : Set the filetype of f to DISKFILE.
            Connect a system's filename (and the file denoted) to the file f
            and open the file for the kind of operation specified in "mode".
            Copy "name" into the "filename" field of f.
 RETURNS  : TRUE if the file was successfully opened, FALSE otherwise.
*)

PROCEDURE putbf(b: BYTE; VAR f : file);
(*
 FUNCTION : write the BYTE b to the file f.
*)

PROCEDURE putcf(c : CHAR; VAR f : file);
(*
 FUNCTION : write the character c to the file f.
*)

PROCEDURE putlf(VAR f : file);
(*
 FUNCTION : write a caracter NEWLINE character to the file f.
*)

PROCEDURE putrf(record : ARRAY OF BYTE; VAR f : file);
(*
 FUNCTION : write a record to file f. The number of BYTEs written is
            determined by the size of the actual parameter.
*)

PROCEDURE putstr(str : ARRAY OF CHAR; VAR f : file);
(*
 FUNCTION : writes the characters in str up to, but not including the
            terminating ENDSTR to the file f.
*)

PROCEDURE remove(VAR f : file);
(*
 FUNCTION : close the file and remove it.
 NOTE:      the file must be open !!! (otherwise it is not associated to
            a var of type file!)
*)


PROCEDURE rename(newname : ARRAY OF CHAR; VAR f : file) : BOOLEAN;
(*
 FUNCTION: rename file f to new name
 RETURNS : TRUE if the file was successfully renamed, FALSE otherwise.
*)


PROCEDURE seek(highpos, lowpos : CARDINAL; mode : IOmode; VAR f : file);
(*
 FUNCTION : positions the file such that a subsequent read or write will
            access the byte number (highpos*MAXCARD+lowpos). Byte numbers
            are counted from the beginning of the file starting with zero.
            Set the IOmode to "mode".
```

```
 PARAMETER: if mode = IOWRITE then the file is cut to the actual position.
            if mode = IOVERRIDE then the file is overwritten from
            the actual position on, the rest of the file remaining unchanged.
            all other modes : the file keeps it's length and is
            set to the specified mode.
 NOTE       : seek can change the IOmode of a file which was opened for another
            IOmode.
 NOTE       : if the actual highpos and lowpos are too big - which corresponds
            to positionning behind the end of the file - an
            error message is displayed and the program is aborted.
 NOTE       : on the SUN, highpos has no meaning (is considered to be 0
            even if another actual value is passed).
*)


PROCEDURE ungetf(VAR f : file);
(*
FUNCTION :  The call of this procedure informs the procedures getbf or getcf
            that the next byte should not be read from the file,
            but a copy of the last read byte should be returned.
*)

END HPfiles.
```

```
DEFINITION MODULE HSconversions;
(* HSconversions ++ string conversions from/to CARDINAL and INTEGER *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHORS:        Michel Kiener, Alfred Ultsch
   $DATE:           February 17th, 1987        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HSconversions.def
   $MODIFICATIONS:
   $DESCRIPTION:    Procedures for conversion of strings to CARDINAL (in any
                    base with 2 <= base <= 36) and INTEGER and vice versa
                    (in bases > 10, upper case letters A..Z are used as
                    digits in the following way:
                    "A" in any base > 10 means "10" in base 10,
                    "B" in any base > 11 means "11" in base 10,
                    ...
                    "Y" in base 35 or 36 means "34" in base 10,
                    "Z" in base 36 means "35" in base 10).
   $NOTE:           For the definition of a string with the maximal
                    number of characters in it, import the type
                    "string" (= ARRAY [0 .. MAXSTR-1] OF CHAR)
                    from HConstypes.
   $NOTE:           For the used conventions about strings, see
                    $DESCRIPTION of DEFINITION MODULE HSfunctions.
   $NOTE:           The procedures exported by this module return only a simple
                    error status.
                    Applications requiring more detailled error informations
                    may get the error number passed to the error procedure.
                    See section 2.6. of the report:
                         HOST: An Abstract Machine for Modula-2 Programs
                         by Michel Kiener and Alfred Ultsch
                         Report of the Institut fuer Informatik der ETH Zuerich
                         February 1987.
*)


(* Alphabetical List of Procedures: *)
(* ------------------------------- *)
(* CardinalToString -- conv. CARD. to str. representing it in any base <= 36  *)
(* cardtoc          -- convert a CARDINAL to a string representing it in base 10  *)
(* ctocard          -- conv. str. representing a cardinal in base 10 to CARDINAL  *)
(* ctoi             -- conv. str. representing an integral in base 10 to INTEGER  *)
(* itoc             -- convert an INTEGER to a string representing it in base 10  *)
(* StringToCardinal -- conv. str. representing card. in base <= 36 to CARDINAL *)



(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED
  (* - to/from CARDINAL in base 10   *) ctocard,          cardtoc,
  (* - to/from CARDINAL in any base  *) StringToCardinal, CardinalToString,
  (* - to/from INTEGER               *) ctoi,             itoc;



(* Description of exported Procedures: *)
(* ----------------------------------- *)

PROCEDURE CardinalToString(number,
                           base :     CARDINAL;
                           position : INTEGER;
```

```
                        minus:      BOOLEAN;
                        VAR s :     ARRAY OF CHAR) : INTEGER;
(*
 FUNCTION : Convert the CARDINAL "number" to a character sequence
            representing it in base "base" (2 <= base <= 36).
            If "minus" = TRUE and "number" # 0, the generated
            character sequence contains a "-" as its first character.
            If the generated character sequence (possibly including
            a "-" character) fits into s (i.e. if its length is
            less than or equal to HIGH(s) - "position" + 1), insert
            it into s with the first character inserted at the place
            of the ("position" + 1)-th character of s.
            If possible (i.e. if s[HIGH(s)] is still unoccupied),
            insert an ENDSTR character into s after the inserted sequence.
            If the generated character sequence does not fit into s,
            nothing is inserted into s, and the procedure returns FAILED.
 RETURNS  : The next position after the insertion (which is either
            occupied by an ENDSTR character or points outside s)
            if the generated character sequence could be inserted
            into s, FAILED otherwise.
 NOTE     : The constant FAILED = -1 is exported by HConstypes
 NOTE     : Returned value > 0      <==> successful conversion
            Returned value = FAILED <==> the generated character sequence
                                         does not fit into the actual string.
 EXAMPLE  : s : ARRAY [0 .. 20] OF CHAR;
            x:=CardinalToString(123, 10, 1, TRUE, s);
            --> s[0] = (unchanged)
                s[1..4] = "-123"
                s[5] = ENDSTR
                x = 5
*)


PROCEDURE cardtoc(c : CARDINAL; VAR s : ARRAY OF CHAR; i : INTEGER):INTEGER;
(*
 FUNCTION : Convert the CARDINAL c to a character sequence representing
            it in base 10. If the generated character sequence
            fits into s (i.e. if its length is less than or equal
            to HIGH(s) - "i" + 1), insert it into s with the first
            character inserted at the place of the ("i" + 1)-th character of s.
            If possible (i.e. if s[HIGH(s)] is still unoccupied),
            insert an ENDSTR character into s after the inserted sequence.
            If the generated character sequence does not fit into s,
            nothing is inserted into s, and the procedure returns
            the value FAILED.
 NOTE:      x:=cardtoc(c, s, i) is equivalent to
            x:=CardinalToString(c, 10, i, FALSE, s).
 RETURNS  : The next position after the insertion (which is either
            occupied by an ENDSTR character or points outside s)
            if the generated character sequence could be inserted
            into s, FAILED otherwise.
 NOTE     : The constant FAILED = -1 is exported by HConstypes
 NOTE     : Returned value > 0      <==> successful conversion
            Returned value = FAILED <==> the generated character sequence
                                         does not fit into the actual string.
 EXAMPLE  : s : ARRAY [0 .. 20] OF CHAR;
            x:=cardtoc(123, s, 0);
            --> s[0 .. 2] = "123"
```

```
                s[3] := ENDSTR
                x := 3
*)



PROCEDURE ctocard(VAR c:CARDINAL; s:ARRAY OF CHAR; VAR i:INTEGER) : BOOLEAN;
(*
 FUNCTION : The string s is scanned from left to right,
            starting from the ("i" + 1)-th character of s.
            Leading blanks and tabs are skipped. Then a string
            representing a cardinal number in base 10 is expected.
            Scanning stops either at the end of the string s, or when
            a character which is not a digit in base 10 is encountered.
            There are two possible cases:
            - the string does not represent a cardinal number in base 10
              or represents a cardinal number greater than MAXCARD:
              then "c" is assigned the value 0, "i" remains unchanged
              and the procedure returns FALSE.
            - the string represents a cardinal number that can be converted:
              then "c" is assigned the value of the converted number, "i"
              points to the next position after the number and the procedure
              returns TRUE.
 NOTE:      x:=ctocard(c, s, i) is equivalent to
            x:=StringToCardinal(s, 10, i, bool, c); IF bool THEN c:=0; x:=0 END;
 RETURNS  : The status of the conversion:
              TRUE:             indicates successful conversion
              FALSE:            indicates that the string does not represent
                                a number in the expected format or that
                                the string represents a syntactically valid
                                number which is outside the machine dependent
                                legal range for that type.
 EXAMPLES:  1)  s : ARRAY [0 .. 20] OF CHAR;
                s[2..7] := "123ABC";
                i := 2;
                x:=ctocard(c, s, i);
                --> c := 123
                    x := TRUE
                    i := 5
            2)  s : ARRAY [0 .. 20] OF CHAR;
                s[0..17] := "999999999999BLABLA";
                i := 0;
                x:=ctocard(c, s, i);
                --> c := 0
                    x := FALSE
                    i := 0
            3)  s : ARRAY [0 .. 20] OF CHAR;
                s[1..3] := "BLA";
                i := 1;
                x:=ctocard(c, s, i);
                --> c := 0
                    x := FALSE
                    i := 1
*)


PROCEDURE ctoi(VAR n: INTEGER;  VAR s: ARRAY OF CHAR; VAR i: INTEGER): BOOLEAN;
```

```
(*
 FUNCTION : The string s is scanned from left to right,
            starting from the ("i" + 1)-th character of s.
            Leading blanks and tabs are skipped. Then a string
            representing an integral number is expected.
            Scanning stops either at the end of the string s, or when
            a character which is not a digit in base 10 is encountered.
            There are two possible cases:
            - the string does not represent an integral number
              or it represents an integral number smaller than MININT
              or greater than MAXINT: then "n" is assigned the value 0,
              "i" remains unchanged and the procedure returns FALSE.
            - the string represents a cardinal number that can be converted:
              then "n" is assigned the value of the converted number,
              "i" points to the next position after the number
              and the procedure returns TRUE.
 RETURNS  : The status of the conversion:
              TRUE:          indicates successful conversion
              FALSE:         indicates that the string does not represent
                             a number in the expected format or that
                             the string represents a syntactically valid
                             number which is outside the machine dependent
                             legal range for that type.
 EXAMPLES:  1)  s : ARRAY [0 .. 20] OF CHAR;
                s[2..7] := "-123AB";
                i := 2;
                x:=ctoi(n, s, i);
                --> n := -123
                    x := TRUE
                    i := 6
            2)  s : ARRAY [0 .. 20] OF CHAR;
                s[1..18] := "999999999999BLABLA";
                i := 1;
                x:=ctoi(n, s, i);
                --> n := 0
                    x := FALSE
                    i := 1
            3)  s : ARRAY [0 .. 20] OF CHAR;
                s[1..3] := "BLA";
                i := 1;
                x:=ctoi(n, s, i);
                --> n := 0
                    x := FALSE
                    i := 1
*)


PROCEDURE itoc(n : INTEGER; VAR s : ARRAY OF CHAR; i : INTEGER) : INTEGER;
(*
 FUNCTION : Convert the INTEGER n to a character sequence representing
            it in base 10. If the generated character sequence
            fits into s (i.e. if its length is less than or equal
            to HIGH(s) - "i" + 1), insert it into s with the first
            character inserted at the place of the ("i" + 1)-th character of s.
            If possible (i.e. if s[HIGH(s)] is still unoccupied),
            insert an ENDSTR character into s after the inserted sequence.
            If the generated character sequence does not fit into s,
            nothing is inserted into s, and the procedure returns
            the value FAILED.
 RETURNS  : The next position after the insertion (which is either
```

```
            occupied by an ENDSTR character or points outside s)
            if the generated character sequence could be inserted
            into s, FAILED otherwise.
 NOTE     : The constant FAILED = -1 is exported by HConstypes
 NOTE     : Returned value > 0      <==> successful conversion
            Returned value = FAILED <==> the generated character sequence
                                         does not fit into the actual string.
 EXAMPLE  : s : ARRAY [0 .. 20] OF CHAR;
            x:=itoc(123, s, 0);
            --> s[0 .. 2] = "123"
                s[3] = ENDSTR
                x = 3
*)

PROCEDURE StringToCardinal(s : ARRAY OF CHAR;
                           base: CARDINAL;
                           VAR position : INTEGER;
                           VAR minus: BOOLEAN;
                           VAR number : CARDINAL) :  BOOLEAN;
(*
 FUNCTION : The string s is scanned from left to right,
            starting from the ("position" + 1)-th character of s.
            Leading blanks and tabs are skipped. Then a string
            representing a cardinal number in base "base"
            (2 <= base <= 36) is expected. If "base" = 10, a sign
            character (plus or minus) in front of the number is accepted.
            Scanning stops either at the end of the string s, or when
            a character which is not a digit in base "base" is encountered.
            There are two possible cases:
            - the string does not represent a cardinal number in base "base"
              or the string represents a cardinal number greater than MAXCARD:
              then "number" is assigned the value 0, "position" remains
              unchanged and the procedure returns FALSE.
            - the string represents (in base "base") a cardinal number
              that can be converted: then "number" is assigned the value
              of the converted number, "position" points to the next
              position after the number and the procedure returns TRUE.
            If    "base" = 10
            and   there is a minus sign in front of the scanned string
            and   the converion is successful
            then  "minus" becomes TRUE.
            In all other cases, "minus" becomes FALSE.
 RETURNS  : The status of the conversion:
              TRUE:           indicates successful conversion
              FALSE:          indicates that the string does not represent
                              a number in the expected format or that
                              the string represents a syntactically valid
                              number which is outside the machine dependent
                              legal range for that type.
 EXAMPLES:  1)  s : ARRAY [0 .. 20] OF CHAR;
                s[2..7] = "12BCXY";
                position = 2;
                x:=StringToCardinal(s, 16, position, minus, number);
                --> number = 4796
                    x = TRUE
                    position = 6
            2)  s : ARRAY [0 .. 20] OF CHAR;
                s[1..18] = "999999999999BLABLA";
                position = 1;
                x:=StringToCardinal(s, 10, position, minus, number);
```

```
                --> number = 0
                    x = FALSE
                    position = 1
            3)  s : ARRAY [0 .. 20] OF CHAR;
                s[1..4] = " BLA";
                position = 1;
                x:=StringToCardinal(s, 10, position, minus, number);
                --> number = 0
                    x = FALSE
                    position = 1
*)


END HSconversions.
```

```
DEFINITION MODULE HSreal;
(* HSreal ++ string conversions from/to REAL *)
(* $HOST-COMPUTER: SUN, Lilith, Macintosh
   $AUTHOR:        Michel Kiener
   $DATE:          February 17th, 1987        $VERSION:    2.0
   $PROJECT:       HOST                       $FILE:       HSreal.def
   $MODIFICATIONS:
   $DESCRIPTION:   Procedures for conversion of strings to REAL and
                   vice versa.
   $NOTE:          For the definition of a string with the maximal
                   number of characters in it, import the type
                   "string" (= ARRAY [0 .. MAXSTR-1] OF CHAR)
                   from HConstypes.
   $NOTE:          For the used conventions about strings, see
                   $DESCRIPTION of DEFINITION MODULE HSfunctions.
   $NOTE:          The procedures exported by this module return only a simple
                   error status.
                   Applications requiring more detailled error informations
                   may get the error number passed to the error procedure.
                   See section 2.6. of the report:
                        HOST: An Abstract Machine for Modula-2 Programs
                        by Michel Kiener and Alfred Ultsch
                        Report of the Institut fuer Informatik der ETH Zuerich
                        February 1987.
*)


(* Alphabetical List of Procedures: *)
(* -------------------------------- *)
(* RealToString    -- convert a REAL to a string representing it          *)
(* StringToReal    -- convert a string representing a real number to a REAL *)


(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED RealToString, StringToReal;


(* Description of exported Procedures: *)
(* ----------------------------------- *)

PROCEDURE RealToString (r:           REAL;
                        i:           INTEGER;
                        VAR s:       ARRAY OF CHAR;
                        length,
                        precision:   CARDINAL;
                        scientific:  BOOLEAN) :          INTEGER;
(*
 FUNCTION : Convert the REAL "r" to a character sequence of "length"
            characters representing it. If "length" is greater than
            required, the generated character sequence is filled up
            with leading spaces. If the generated character sequence
            fits into s (i.e. if "length" is less than or equal to
            HIGH(s) - "i" + 1), insert it into s with the first
            character inserted at the place of the ("i" + 1)-th
            character of s. If possible (i.e. if s[HIGH(s)] is still
            inoccupied), insert an ENDSTR character into s after
            the inserted sequence (i.e. into the ("i" + "length" + 1)
            character of s). If "length" is smaller than the number of
            characters needed to represent the number or if the
            generated character sequence does not fit into s, nothing
```

```
            is inserted into s (i.e. s remains unchanged) and the
            procedure returns FAILED.
            Otherwise the procedure returns the index of
            the next position after the insertion (i.e. the index of
            the ("i" + "length" + 1)-th character of s - which is
            either occupied by an ENDSTR character or "lies outside" s -).

            If "scientific" = FALSE, the argument is converted to
            decimal notation of the format:

            real = ["-"] digits "." digits

            digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

            digits = digit {digit}

            where the number of digits after the decimal point is specified
            by "precision".

            If "scientific" = TRUE, the resulting string has the formrat:

            real = ["-"] digit "." digits "E" ["+"|"-"] digit digit

            where the digit before the decimal point is the first
            significant digit, the number of digits between "." and "E"
            is specified by "precision" and the exponent is represented
            by two digits preceeded by a plus or a minus sign.
            If the given "length" is too small for the given "precision",
            nothing is inserted into s and the procedure returns the
            value 0.
 RETURNS  : if the conversion was successful, ("i" + "length") is returned,
            - i.e. the index of the next position after the insertion
            (which is either occupied by an ENDSTR character or "lies
            outside" s) -.
            if the conversion was not successful, FAILED is returned.
 NOTE     : The constant FAILED = -1 is exported by HConstypes
 NOTE     : Returned value > 0      <==> successful conversion
            Returned value = FAILED <==> the generated character sequence
                                         does not fit into the actual string.

 EXAMPLES:  1)  s : ARRAY [0 .. 20] OF CHAR;
                x:=RealToString(123.45, 1, s, 5, 1, FALSE);
                --> s[0] = (unchanged)
                    s[1..5] = "123.4"
                    s[6] = ENDSTR
                    x = 6
            2)  s : ARRAY [0 .. 20] OF CHAR;
                x:=RealToString(123.45, 1, s, 2, 1, FALSE);
                --> s[0 .. 20] = (unchanged)
                    x = FAILED
            3)  s : ARRAY [0 .. 20] OF CHAR;
                x:=RealToString(-1.5, 0, s, 10, 3, TRUE);
                --> s[0 .. 9] = "-1.500E+00"
                    s[10] = ENDSTR
                    x = 10
*)


PROCEDURE StringToReal (s: ARRAY OF CHAR;
                        VAR r: REAL;
                        VAR i: INTEGER) : BOOLEAN;
```

```
(*
 FUNCTION : The string s is scanned from left to right,
            starting from the ("i" + 1)-th character of s.
            Leading blanks and tabs are skipped. Then a string
            representing a real number in the format:

            real = ["+"|"-"] digits ["." digits] ["E" ["+"|"-"] digits]

            digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

            digits = digit {digit}

            is expected. Scanning stops either at the end of the string s,
            or when a character which does not fit in the format is encountered.
            There are two possible cases:
            - the string does not represent a real number or it
              represents a syntactically valid real number
              which is outside the machine defined range for the type
              REAL: then "r" is assigned the value 0.0, "i" remains
              unchanged and the procedure returns FALSE.
            - the string represents a syntactically valid real number
              that can be converted: then "r" is assigned the value
              of the converted number, "i" points to the next
              position after the number and the procedure returns
              TRUE.
 RETURNS  : The status of the conversion:
              TRUE:            indicates successful conversion
              FALSE:           indicates that the string does not represent
                               a number in the expected format or that
                               the string represents a syntactically valid
                               number which is outside the machine dependent
                               legal range for that type.
 EXAMPLES:  1)  s : ARRAY [0 .. 20] OF CHAR;
                s[0..6] = "XX0.123AB";
                i = 2;
                x:=StringToReal(s, r, i);
                --> r = 0.123
                    x = TRUE
                    i = 7
            2)  s : ARRAY [0 .. 20] OF CHAR;
                s[0..3] = "BLAB";
                i = 1;
                x:=StringToReal(s, r, i);
                --> r = 0.0
                    x = FALSE
                    i = 1
*)

END HSreal.
```

```
DEFINITION MODULE HDwindows;
(* HDwindows ++ screen, window, menue and CARET-handling *)
(* $HOST-COMPUTER:  Lilith, SUN, MacIntosh
   $AUTHOR:         Alfred Ultsch
   $DATE:           July 14th, 1986          $VERSION:    1.3
   $PROJECT:        HOST                     $FILE:       HDwindows.def
   $MODIFICATIONS:
   $DESCRIPTION:
   This module defines operations for the following objects:

   Screen: A rectangular display area dimensioned in pixel-units
           (x,y-coordinates) painted with a BACKGROUND-colour. A screen can
           be opened, cleared (set to a background colour) and closed.
           A screen has a fix-dimentioned screen-window (FIRSTWINDOW) and a
           BACKGROUND-menue. The screens' window can have a title and a menue.
           On a screen a set of 0 to MAXWINDOW windows can be
           simultaneously displayed with evental BACKGROUND in between them.
           At one time there is one CURSOR to be seen somewhere
           on the screen and a CARET marks a position in some windows.
           If the CURSOR is pointing to the BACKGROUND and a MENUE (see below)
           is requested, a simple menue, that allows the change of
           window-boundaries is displayed.
           If the standard-output STDOUT was not previously
           redirected to a file, it is redirected to the screen-window.

   Window: A rectangular area dimensioned in pixel-units (x,y-coordinates)
           that shows (a part of) a text stream (file).To every window belongs
           eventually a title (line of text) and a location bar.
           When a window is opened through a call of
           HDwindows.openw(name, ..., window, windownumber),
           the actual "name" is copied into the "filename"-field ofthe file
           variable "window", so that it can be asked later.
           From then on the window boundaries may be changed by a hidden
           mechanism if the window is opened RESIZABLE. If the window is
           opened NOTRESIZABLE then the window's location and boundaries
           remain static as long as the window is open.
           i.e. the number of lines/colums displayed can vary. Each time the
           window's dimensions are changed (and also at opening time), a
           restore-procedure is called. The actual line/column-dimensions can
           be inquired any time. Writing starts at the top-left corner,
           writing is done at the CARET's position. The window wraps around
           to the beginning of the next line when the right margin is reached.
           At most one menue is associated with a window. Menues are used to
           commands. The command selection is done by a hidden mechanism but
           can be inquired by calling procedure command. Since command
           selection is real time, command should be called frequently (busy
           read). The same procedures for writing into a file (putbf, putcf,
           putlf, putrf, seek) are used for writing to a window.

           Interpretation of characters:
           When writing characters to a window, the following interpretations
           are done (CARET is synonymous for the actual writing position):
           10C  BS  backspace CARET without deletion
           12C  LF  move CARET to next line,same column
           14C  FF  clear window,  CARET to top-left
           15C  CR  move CARET to same line,first column
           30C  CAN clear the line where CARET is, CARET to the same line
                    first column
           34C  FS  write special caracter for end-of-file.
           35C  GS  write special caracter G/S
           36C  EOL move CARET to next line, first column.
```

```
        37C  US  write special caracter U/S
       177C  DEL delete caracter, backspace one position (exept at the top)
        The NEWLINE-character separates the byte-streams into different
        lines.
        Other characters < 37C (US) have the effect of deleting anything
        at the CARET's actual position.


   Cursor: A pointing mark somewhere on the screen. The cursor is moved on the
           screen by some input mechanisms, that is hidden in the
           implementation (maybe a roll-balled mouse).
           The cursor movements are actualized by calling command. At a given
           time (for example when a BUTTON on the button-device is pressed),
           its position with respect to a window can be inquired.
           The cursor's position is given in coordinates of lines (starting
           from the  topline as number 1) and bytes (starting from the
           leftmost as number 1) relative to the window where the cursor is
           in. Special positions of the cursor and pressing of buttons have
           the effect of selecting a command (see below) or moving a window
           boundary (see above).

   Menues: At most one menue can be seen on a screen at a time.
           A menue is used to select commands this selection is done in real
           time. When a menue entry is selected, the user-redefinable
           procedure (variable) MenueRequest is called.
           Hidden in the implementation are the details of menues'
           pop-up and disappearance as well as the details of command
           selection.

   Caret:  A caret marks a position in each window.
           (for example to mark a position to the left of which the next
           character is inserted when keyboard input is done).
           A caret can be positioned (or its position inquired) in coordinates
           of lines (starting from the topline as number 1)
           and bytes (starting from the leftmost as number 1) relative to
           the window where the caret is in.
           The caret's position can be inquired any time.

   $NOTE:  The procedures exported by this module return only a simple
           error status.
           Applications requiring more detailled error informations
           may get the error number passed to the error procedure.
           See section 2.5. of the report:
                       HOST: An Abstract Machine for Modula-2 Programs
                       by Michel Kiener and Alfred Ultsch
                       Report of the Institut fuer Informatik der ETH Zuerich
                       February 1987.
*)


(* Alphabetical List of Procedures: *)
(* ------------------------------- *)
(* clearcaret   -- turn the display of the CARET off                        *)
(* clearscr     -- close all open windows reset the colour to background *)
(* closescr     -- close a screen reset STDOUT(eventually)                  *)
(* closew       -- close a text window                                      *)
(* command      -- look for menue- commands                                 *)
(* dimensions   -- return the actual window's site and dims                 *)
(* getcaretpos  -- get the current position of the CARET                    *)
(* linecol      -- return the actual window's lines/columns                 *)
(* locatecursor -- find out where the CURSOR is on the screen               *)
```

```
(* mark          -- highlight a character of a window                    *)
(* openscr       -- open a screen,display name,redefine STDOUT(eventually)*)
(* openw         -- open text window                                     *)
(* scrollup      -- scroll window content up one line                    *)
(* setcaret      -- position the CARET inside a specific window          *)



(* Import List: *)
(* ------------ *)
FROM HConstypes IMPORT (* Type  *) file;



(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED
(* Constants:  *) BACKGROUND, FIRSTWINDOW, MAXWINDOW,
(* Types:      *) WINDOWNR, WINDOWPROC, WINDOWSIZE,
(* Variables:  *) MenueRequest, Screen,
(* Procedures: *)
(* screen           *) openscr, closescr, clearscr,
(* window           *) openw,   closew,   dimensions, linecol, scrollup,
(* menue            *) command,
(* cursor           *) locatecursor,
(* CARET            *) setcaret, getcaretpos, clearcaret,
(* mark coordinates *) mark;



(* Declaration of exported Cpnstants, Types and Variables: *)
(* ------------------------------------------------------- *)
CONST
 BACKGROUND  = 0; (* "windownumber" of screen area where no window is *)
 FIRSTWINDOW = 1; (* the smallest window-identifying number *)
 MAXWINDOW   = 8; (* the largest window-identifying number = max. nr. of
                     windows *)


TYPE WINDOWNR = [BACKGROUND..MAXWINDOW]; (* possible windownumbers *)
TYPE WINDOWPROC = PROCEDURE(WINDOWNR);
                (* The type for the procedures to refresh a window's content.
                    WINDOWNR : what window is to refresh
                *)
TYPE WINDOWSIZE =  (RESIZABLE,NOTRESIZABLE); (* see descrption of screen
                                                above *)

VAR
   Screen : file ;   (* if a screen is opened, then associated to the screens'
                        basic window, else synonymous to STDOUT *)
   MenueRequest : PROCEDURE(WINDOWNR, CARDINAL);
   (* This procedure is called whenever a menue appeared on the screen.
      Parameters are:
      windownumber : WINDOWNR : The number of the window where the menue
                                request was
      menueEntryNumber : CARDINAL : The menuenumber is interpreted as octal
                                    number.
      If no menue entry was selected  menueEntryNumber is zero. Each
```

```
      octal position corresponds to a submenue level.The rightmost
      octal number indicates the command chosen in the main menue,
      the digit at the next higher position indicates the commmand
      in the first submenue etc. The initial value determines
      the inital selection when the menue pops up.
*)



(* Description of exported Procedures: *)
(* ---------------------------------- *)

(*--------------------  PROCEDURES for SCREEN handling ------------------*)

PROCEDURE openscr(name : ARRAY OF CHAR; menue : ARRAY OF CHAR;
                  VAR width, height : CARDINAL) : BOOLEAN;
(*
 PARAMETER:
   name           : if the name is nonempty a title-line is dispalyed
   width, height  : screen dimensions in pixel-unit's
   menue          : the description of a menue that can be called on the
                    screen's FIRSTWINDOW.
 FUNCTION : Opens up a rectangular display area, returns it dimensions
            in pixel-unit's. Furthermore, FIRSTWINDOW is opened
            with fixed dimensions and men "menue".
            IF STDOUT was not redirected to a file, all put's go to
            the screen's FIRSTWINDOW. NOTE: this can be changed by simply
            assigning (:=) another file to STDOUT.
 RETURNS  : TRUE if the screen could be opened
*)

PROCEDURE clearscr ();
(*
 FUNCTION : all open windoes are closed, the scren is reset to the same
            condition as after openscr
*)

PROCEDURE closescr ();
(*
 FUNCTION : all open windows are closed, STDOUT is reset if it was redefined
*)

(*--------------------  PROCEDURES for WINDOW handling ------------------*)

PROCEDURE openw(name : ARRAY OF CHAR;
                menue : ARRAY OF CHAR;
                x,
                y,
                width,
                height : CARDINAL;
                resizable : WINDOWSIZE;
                content : WINDOWPROC;
                VAR window : file;
                VAR windownumber : WINDOWNR) : BOOLEAN;
(*
   INPUT PARAMETERS:
   name           : if the name is nonempty a title-line is drawn
   x, y           : window coordinates in pixel-unit's starting down, left
   width, height  : window dimensions  in pixel-unit's
```

```
   resizable      : if the size and location of the window may be changed
   content        : a procedure (e.g. one that is able to write the whole window
                    content at a time) which is called once at opening time,
                    then whenever the window's dimensions are changed
   menue          : the description of a menue
   OUTPUT PARAMETERS:
   window         : the opened window as file
   windownumber   : a unique number to identify the window in the range
                    from FIRSTWINDOW to MAXWINDOW.
 FUNCTION : Opens a 'window' in the form of a IOWRITE file.
            All file-writing procedures are applicable.
            The CARET is set to the top left of the writable field.
 RETURNS  : TRUE iff the window was successfully opened.
*)


PROCEDURE closew(windownr : WINDOWNR);
(*
 FUNCTION : close the window, reduce the occupied space to BACKGROUND
*)


PROCEDURE dimensions(windownumber: WINDOWNR; VAR x,y,width,height: CARDINAL);
(*
 PARAMETER: the window for which the dimensions are asked
 FUNCTION :  returns the actual window location and dimensions in pixels
*)

PROCEDURE linecol(windownumber : WINDOWNR; VAR lines, colums : CARDINAL);
(*
 PARAMETER: the window for which the dimensions are asked
 FUNCTION : returns the actual window dimensions in lines and colums
*)

PROCEDURE mark(windownumber : WINDOWNR; line, byte : CARDINAL) : BOOLEAN;
(*
 FUNCTION : the specified position is marked visible (inverted or so).
            A second call with the same coordinates will reset the
            caracter to normal looks.
 RETURNS  : TRUE if the marking could be done.
*)

PROCEDURE scrollup(windownumber: WINDOWNR);
(*
 FUNCTION : scroll the window up one line
*)

(*-------------------- PROCEDURES for meue handling ------------------*)

PROCEDURE command() : BOOLEAN;
(*
 FUNCTION : Inquire if a command of a certain menue is actually pressed.
            NOTE:  The command seletion is time-dependend (busy-read), so
            it's usefull to call this procedure frequently.
            As side effects this procedure may:
            1) Adjust the cursor according to user-input
            2) Let menues pop-up and disappear
            3) Let window-boundaries be changed
            If the user selects a menue-command, the MenueRequest perform
            is called.
 RETURNS  : TRUE if the display of a menue has been requested.
```

```
*)

(*--------------------  PROCEDURES for CURSOR handling -----------------*)

PROCEDURE locatecursor(VAR windownumber : WINDOWNR;
                       VAR line,
                           byte : CARDINAL)              : WINDOWNR;
(*
 FUNCTION : Gives the window and the coordinates, where the cursor is at the
            moment
 RETURNS  : The windownumber where the cursor is
*)

(*--------------------  PROCEDURES for CARET handling -----------------*)

PROCEDURE setcaret(windownumber: WINDOWNR; line, byte : CARDINAL) : BOOLEAN;
(*
 PARAMETER:
  windownumber : the window where the caret is set. If this
                 number = BACKGROUND then the caret is turned off.
  line : the line on which the caret is set in the window, starting from the
         top line as number one.
  byte : the caret's position in the line in number of displayed bytes
         starting from the leftmost byte as number one.
 FUNCTION : positions the caret to a location inside a window.
 RETURNS  : TRUE if the positioning could be done.
*)

PROCEDURE clearcaret();
(*
 FUNCTION : switches the display of the CARET off.
            If no CARET was displayed, the procedure has no effects.
*)

PROCEDURE getcaretpos(VAR windownumber : WINDOWNR; VAR line, byte : CARDINAL);
(*
 PARAMETER:
  windownumber : the window where the caret is currently in or
                 BACKGROUND if the carret is not on the screen
  line : the line on which the caret is in the window, staring from the
         top line as number one.
  byte : the caret's position in the line in number of displayed bytes
          starting from the leftmost byte as number one.

 FUNCTION : Inquire the caret's actual position.
*)

END HDwindows.
```

```
DEFINITION MODULE HDbutton;
(* HDbutton ++  input device with button(s) *)
(* $HOST-COMPUTER:  Lilith (other versions exist for SUN and Macintosh)
   $AUTHOR:         Alfred Ultsch
   $DATE:           July 14th, 1985          $VERSION:    1.3
   $PROJECT:        HOST                     $FILE:       HDbutton.def
   $MODIFICATIONS:
   $DESCRIPTION:    This module defines an input device with several buttons:
                    One or more of these buttons can be pressed at any time.
                    If one of the BUTTONs on the button-device is pressed,
                    the procedure pressed returns a BITSET enclosing
                    a constant corresponding to that button.
                    Inquiring the state of the buttons is real time (busy-read).
                    The buttons can, but must not, have a special meaning
                    in connection with windows and the CURSOR.
*)

(* Alphabetical List of Procedures: *)
(* ------------------------------- *)
(* pressed       -- inquire state of buttons *)


(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED
(* Constants: *) BUTTONNUMBER, POINTINGBUTTON, MENUEBUTTON, MARKINGBUTTON,
(* Type:      *) BUTTONPROC,
(* Procedure: *) pressed;


(* Declaration of exported Constants: *)
(* ---------------------------------- *)
CONST  BUTTONNUMBER   = 3; (* how many buttons there are *)
       POINTINGBUTTON = 15;
       MENUEBUTTON    = 14;
       MARKINGBUTTON  = 13;
       (* These names of the BITSET elements for the LILITH-mouse BUTTONs
          may suggest a certain meaning in the design of a user interface *)


(* Declaration of exported Type: *)
(* ----------------------------- *)
TYPE BUTTONPROC = PROCEDURE(BITSET) : BOOLEAN; (* type of PROCEDURE pressed *)



   Description of exported Procedure: *)
   ---------------------------------- *)
PROCEDURE pressed(VAR button : BITSET) : BOOLEAN;
*
FUNCTION : If a button is pressed the appropriate BUTTON is
           a member of the returned BITSET.
           The procedure returns immediately, even when no BUTTON is pressed.
           NOTE: the button state is checked real-time(busy read).
RETURNS  : TRUE if any button is actually pressed
*)


END HDbutton.
```

```
DEFINITION MODULE HDbar;
(* HDbar  ++  scroll bars with optional rectangles        *)
(* $HOST-COMPUTER:  SUN, Lilith, MacIntosh
   $AUTHORS:        Martin Ester, Alfred Ultsch
   $DATE:           July 14th, 1986          $VERSION:    1.3
   $PROJECT:        HOST                     $FILE:       HDbutton.def
   $DESCRIPTION:    This module defines the installation of a bar to a window.
                    A scroll bar is a retangular area at the left
                    side of a window. Pressing a mouse-button inside a
                    scroll bar results in a call to the procedure
                    barcommand, which is exported as procedure variable by
                    this module in order to supply user-defined reactions.
                    Installed scroll bars can be enriched by a background
                    colored rectangle which can be dimensioned in per-mille
                    of the total bar length. A scroll bar (incl. rectangle)
                    is always painted and refreshed by a hidden mechanism
                    (even after changes of the outlook of the window).
   $NOTE:           The procedures exported by this module return only a simple
                    error status (TRUE indicating success and FALSE failure).
                    Applications requiring more detailled error informations
                    may get the error number passed to the error procedure.
                    See section 2.5. of the report:
                        HOST: An Abstract Machine for Modula-2 Programs
                        by Michel Kiener and Alfred Ultsch
                        Report of the Institut fuer Informatik der ETH Zuerich
                        February 1987.


*)



(* Alphabetical List of Procedures: *)
(* ------------------------------- *)
(* deletebar -- remove bar form window                    *)
(* eraserectangle -- let rectangle in a bar dissapear     *)
(* installbar   -- install a scroll bar for a window      *)
(* setrectangle -- position the rectangle of a scrollbar *)



(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED
(* Type:        *) PERMILLE,
(* Variable:    *) barcommand,
(* Procedures: *) deletebar, eraserectangle, installbar, setrectangle;



   Declaration of exported Type: *)
(* ----------------------------- *)
TYPE PERMILLE = [0..1000];



(* Declaration of exported Variable: *)
(* --------------------------------- *)
VAR barcommand : PROCEDURE (VAR CARDINAL, VAR BITSET, VAR PERMILLE);
(* This procedure is called, whenever a button was pressed
   and released inside the region of an installed bar. A user defined
```

```
   procedure may be assigned by the user. The default procedure has no
   |effects.
   Parameters are:
   VAR windownr: CARDINAL the window where the button was released inside
                          the bar's area
   VAR button:   BITSET   an set containing the pressed button(s)
                          see HDbutton for more details
   VAR relpos: PERMILLE   relative position counted from the top of the
                          window where the cursor was at the time of the call.
*)


(* Description of exported Procedures: *)
(* ---------------------------------- *)
PROCEDURE deletebar(windownr : CARDINAL);
(*
   FUNCTION : Remove an installed bar
*)

PROCEDURE eraserectangle(windownr : CARDINAL);
(*
   FUNCTION : If a bar with rectangle was installed, the rectangle dissapears
*)

PROCEDURE installbar(windownr : CARDINAL) :  BOOLEAN;
(*
   FUNCTION : Install a rectangle-less bar for a specific window
   RETURNS  : TRUE iff the bar could be installed
*)

PROCEDURE setrectangle(windownr: CARDINAL; begin, length: PERMILLE) : BOOLEAN;
(*
   PARAMETER: begin defines the begin counted from the top,
              length the relative length of the rectangle
   FUNCTION : the rectangle of bar windownr is set to new values begin
              and length and painted accordingly
   RETURNS  : TRUE iff the setting was successfull
*)

END HDbar.
```

```
DEFINITION MODULE HEreal;
(* HEreal ++ conversions from REAL to INTEGER and vice versa *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHOR:         Michel Kiener
   $DATE:           February 17th, 1987        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HEreal.def
   $MODIFICATIONS:
   $DESCRIPTION:    Procedures for conversions from REAL to INTEGER and
                    vice versa.
*)



(* Alphabetical List of Procedures: *)
(* ------------------------------- *)
(* entier    -- convert a REAL to an INTEGER      *)
(* real      -- convert an INTEGER to a REAL      *)




(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED (* Procedures: *) entier, real;



(* Description of exported Procedures: *)
(* ----------------------------------- *)


PROCEDURE entier(x: REAL): INTEGER;
(*
 FUNCTION : convert a REAL to an INTEGER.
            This procedure is not defined for results that are not in
            the legal INTEGER range.
 RETURNS  : the INTEGER value of the REAL
*)


PROCEDURE real(x: INTEGER): REAL;
(*
 FUNCTION : convert an INTEGER to a REAL
 RETURNS  : the REAL value of the INTEGER
*)

END HEreal.
```

```
DEFINITION MODULE HEmathlib;
(* HEmathlib ++ basic mathematical functions *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHOR:         N. Wirth / J. Waldvogel
   $DATE:           December 10th, 1980        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HEmathlib.def
   $DESCRIPTION:    Basic mathematical functions
*)


(* Alphabetical List of Procedures: *)
(* ------------------------------- *)
(* arctan     -- arc tangent                                  *)
(* sqrt       -- square root                                  *)
(* exp        -- exponential function                         *)
(* ln         -- natural logarithm                            *)
(* cos        -- cosine                                       *)
(* sin        -- sine                                         *)

(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED
(* Constant:   *) pi,
(* Procedures: *) sqrt, exp, ln, sin, cos, arctan;


(* Declaration of exported Constant: *)
(* --------------------------------- *)

CONST pi = 3.1415927;


(* Description of exported Procedures: *)
(* ----------------------------------- *)

PROCEDURE  arctan (x: REAL): REAL;

PROCEDURE  cos (x: REAL): REAL;

PROCEDURE  exp (x: REAL): REAL;

 ROCEDURE  ln (x: REAL): REAL;

 ROCEDURE  sin (x: REAL): REAL;

/ROCEDURE  sqrt (x: REAL): REAL;


END HEmathlib.
```

```
DEFINITION MODULE HEsegmentIO;
(* HEsegmentIO ++ Segment access to mass storage *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHOR:         Michel Kiener and Heinrich Jasper,
                    inspired by SIBlockIO by E. S. Biagioni, G. Heiser,
                                             K. Hinrichs and C. Muller
   $DATE:           February 17th, 1987      $VERSION:   2.0
   $PROJECT:        HOST                     $FILE:      HEsegmentIO.def
   $MODIFICATIONS:
   $DESCRIPTION:    HEsegmentIO:
                    ------------
                    - This module provides low level segment I/O to mass storage
                      (probably on disk).
                    Blocks:
                    -------
                    - Blocks are arrays of bytes of the fixed length BYTESPERBLOCK.
                    Segments:
                    ---------
                    - Integral numbers of blocks grouped into so-called segments
                      may be stored to / retrieved from so-called BIOFiles.
                      From the programmer's point of view, segments are
                      contiguous pieces of memory on the BIOFiles.
                    BIOFiles:
                    ---------
                    - BIOFiles must be opened through a call to OpenBIOFile.
                      This procedure returns a record of type "BIOFile"
                      which contains a unique identifier of the BIOFile: BIOFId.
                      The BIOFId is used in all further block I/O operations
                      on that particular BIOFile and the segments on the BIOFile.
                    - Files of type "BIOFile" are incompatible with the files
                      of type "file" described in the other modules of
                      HOST (HConstypes, HPfiles, HDwindows).
                    - When created (through OpenBIOFile), BIOFiles get a stamp
                      that marks them as BIOFiles. Applying procedures
                      exported by HEsegmentIO to files that do not have that
                      stamp produces an error message on STDERR.
                    Segments again:
                    ---------------
                    - Each segment on a particular BIOFile is uniquely
                      identified by a SegmId which serves as the programmer's
                      address of the segment.
                    - Segments are allocated / deallocated using AllocateSegment /
                      DeallocateSegment and I/O operations are performed on
                      them using PutSegment / GetSegment.
                    - It is the programmer's sole responsiblity to manage her
                      segments, that is to remember and properly use their
                      identifiers (SegmId) and sizes (NrOfBlocks).
                    - For this purpose, HEsegmentIO provides for the usage of
                      so-called SpecialBlocks, which are stored at a particular
                      place of the external BIOFile. A SpecialBlock is read from
                      the external BIOFile when it is opened and written to
                      the external BIOFile when it is closed. Thus, the
                      programmer might use the SpecialBlock to store / retrieve
                      information about identifiers (SegmId) and
                      sizes (NrOfBlocks) of segments onto / from an external
                      BIOFile. The procedure PutSpecialBlock which enables
                      to write at any time a BIOFile's SpecialBlock onto
                      the external BIOFile is provided for data security reasons
                      (e.g. the programmer wishes an updated version
                      of a BIOFile's SpecialBlock to be written externally
                      immediately and not only when the BIOFile is closed).
```

```
                    - The programmer is responsible for remembering the size
                      (NrOfBlocks) of a segment and for using the same size
                      in each procedure call. Failure to observe this rule
                      may result in serious disk and system errors.
                    WaitBIO:
                    --------
                    - HEsegmentIO input / output operations may be implemeted
                      asynchronously, provided the underlying system supports
                      asynchronous I/O.
                    - That requires some care on the programmer's side: a call
                      to GetSegment will only initiate the physical I/O operation.
                      Before accessing the data read into the programmer-supplied
                      buffer, the programmer must await completion of the physical
                      read operation. This is done by a call to the WaitBIO
                      procedure. Similarly, a call to WaitBIO must be performed
                      after calling PutSegment if the programmer-supplied buffer
                      is to be modified afterwards. Of course, the call to
                      WaitBIO does not have to be performed immediately
                      afterwards, the programmer can do other processing in
                      between, thus possibly avoiding any delays due to waiting
                      for I/O completion.
                    - There is an implicit synchronisation done by the system
                      before any further I/O operation and CloseBIOFile are
                      performed on the respective BIOFile. Hence an explicit
                      call to WaitBIO is not required, if a GetSegment
                      operation follows a PutSegment call or vice versa,
                      or between successive calls to GetSegment or PutSegment
                      using different buffers.
   $NOTE:             On some implementations, it might be possible
                      to read sequentially files of type "BIOFile" with
                      the procedures exported by HPfiles (this probably might
                      make sense for test purposes only!).
   $NOTE:             The procedures exported by this module return only a simple
                      error status (TRUE indicating success and FALSE failure).
                      Applications requiring more detailled error informations
                      may get the error number passed to the error procedure.
                      See section 2.5. of the report:
                        HOST: An Abstract Machine for Modula-2 Programs
                        by Michel Kiener and Alfred Ultsch
                        Report of the Institut fuer Informatik der ETH Zuerich
                        February 1987.
   $NECESSARY CONDITIONS: Opened BIOFiles must be closed before program
                      termination.
*)


   Alphabetical List of Procedures: *)
 * ------------------------------- *)
(* AllocateSegment     -- allocate a segment of memory on a BIOFile      *)
(* BIOFileExists       -- check wether a BIOFile exists                  *)
(* CloseBIOFile        -- close a BIOFile                                *)
(* DeallocateSegment   -- deallocate a segment of memory on a BIOFile    *)
(* GetSegment          -- read a segment of memory on a BIOFile          *)
(* OpenBIOFile         -- open a BIOFile for a specified access          *)
(* PutSegment          -- write a segment of memory on a BIOFile         *)
(* RemoveBIOFile       -- remove a BIOFile                               *)
(* PutSpecialBlock     -- write the special block of a BIOFile           *)
(* WaitBIO             -- wait for completion of previous I/O operation  *)
```

```
(* Import List: *)
(* ------------ *)
FROM HConstypes IMPORT (* CONST *) BYTESPERBLOCK,
                       (* TYPE *)  BYTE;


(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED
(* Types: *)
   Block,
   BIOFile,
(* Procedures: *)
   BIOFileExists, RemoveBIOFile,
   OpenBIOFile, CloseBIOFile,
   AllocateSegment, DeallocateSegment,
   PutSegment, GetSegment,
   PutSpecialBlock,
   WaitBIO;


(* Description of exported Types: *)
(* ------------------------------ *)
TYPE
   Block = ARRAY [0..BYTESPERBLOCK-1] OF BYTE;
   SegmentId = INTEGER;  (* Identifier and programmer's address of a segment *)
   BIOFileId;            (* Identifier of a BIOFile; opaque type *)
   BIOFile = RECORD
               BIOFId      : BIOFileId; (* A unique identifier assigned to
                                           each BIOFile when it is created.
                                           Must be used in all further
                                           operations on that BIOFile. *)
               SpecialBlock : Block;    (* When OpenBIOFile is called,
                                           this block is read from the
                                           external BIOFile. When CloseBIOFile
                                           is called, this block is written
                                           to the external BIOFile.
                                           The SpecialBlock may be used to
                                           store informations about the BIOFile.
                                           Its usage is under the sole
                                           programmer's responsibilty. *)
             END;


(* Description of exported Procedures: *)
(* ----------------------------------- *)


PROCEDURE AllocateSegment(BIOFId        : BIOFileId;
                          NrOfBlocks    : CARDINAL;
                          VAR SegmId    : SegmentId) : BOOLEAN;
(*
 FUNCTION:  Return a SegmId for NrOfBlocks contiguous free blocks.
 PARAMETER: BIOFId:     BIOFileId of the BIOFile on which the segment
                        shall be allocated.
            NrOfBlocks: Number of blocks to allocate for the segment.
            SegmId:     Client's address of the segment.
 RETURNS:   TRUE if a segment of NrOfBlocks blocks could be allocated,
            FALSE otherwise.
*)
```

```
PROCEDURE BIOFileExists(filename : ARRAY OF CHAR) : BOOLEAN;
(*
 FUNCTION:  Check wether a BIOFile with name "filename" exists.
 PARAMETER: filename:  name of the external BIOFile.
 RETURNS:   TRUE if a file named "filename" exists and it is a BIOFile
            (i.e. it has been created through a call of OpenBIOFile).
            FALSE otherwise.
*)


PROCEDURE CloseBIOFile(VAR f : BIOFile) : BOOLEAN;
(*
 NOTE:      f is a VAR parameter for efficiency reasons only.
 FUNCTION:  Close the BIOFile and write its SpecialBlock into the external file
            with the name "filename" given for f when it was opened.
 PARAMETER: f: the BIOFile to close.
 RETURNS:   TRUE if the BIOFile could be closed.
            FALSE if the BIOFId-field of f did not designate an open BIOFile.
 NOTE:      Open BIOFiles must be closed before program termination.
*)


PROCEDURE DeallocateSegment(BIOFId     : BIOFileId;
                            NrOfBlocks : CARDINAL;
                            SegmId     : SegmentId) : BOOLEAN;
(*
 FUNCTION:  Delete the block set identified by SegmId.
 PARAMETER: BIOFId:     BIOFileId of the BIOFile from which the segment
                        shall be deallocated.
            NrOfBlocks: Number of blocks of the segment to deallocate.
                        Must be the same as the NrOfBlocks used for
                        the corresponding call of AllocateSegment.
            SegmId:     Client's address of the segment to delete
 RETURNS:   TRUE if deallocation successful, FALSE otherwise.

*)

PROCEDURE GetSegment(BIOFId      : BIOFileId;
                     NrOfBlocks  : CARDINAL;
                     SegmId      : SegmentId;
                     VAR Segment : ARRAY OF Block) : BOOLEAN;
(*
 FUNCTION:  Initiate reading NrOfBlocks blocks identified by SegmId.
 PARAMETER: BIOFId:     BIOFileId of the BIOFile from which the segment
                        shall be read.
            NrOfBlocks: Number of blocks of the segment to read
                        Must be less than or equal to the NrOfBlocks
                        declared in AllocateSegment and less than or
                        equal to HIGH(Segment) + 1
            SegmId:     Client's address of the segment to read
            Segment:    Read segment
 NOTE:      The segment to be read must have been written before.
 RETURNS:   TRUE if successful, FALSE otherwise.
*)


PROCEDURE OpenBIOFile(filename  : ARRAY OF CHAR;
                      ReadOnly  : BOOLEAN;
                      VAR IsNew : BOOLEAN;
```

```
                        VAR f      : BIOFile) : BOOLEAN;
(*
 FUNCTION:  Open a BIOFile for the specified access
            (ReadOnly = TRUE --> read only, ReadOnly = FALSE --> read and write).
 PARAMETER: filename:  name of the external BIOFile to open
            ReadOnly:  If TRUE:
                         - If a file named "filename" exists, and it is a
                           BIOFile, it is opened for read access only.
                           If opening is successful, the procedure returns
                           TRUE, otherwise FALSE. An attempt to write on
                           a BIOFile opened with ReadOnly = TRUE will
                           generate an error message on STDERR and abort
                           the program.
                         - If a file named "filename" exists, and it is not a
                           BIOFile, it is NOT opened and the procedure
                           returns FALSE.
                         - If no file named "filename" exists, the procedure
                           returns FALSE.
                       If FALSE:
                         - If a file named "filename" exists, and it is a
                           BIOFile, it is opened for read and write access.
                           If opening is successful, the procedure returns
                           TRUE, otherwise FALSE.
                         - If a file named "filename" exists, and it is not a
                           BIOFile, it is NOT opened and the procedure
                           returns FALSE.
                         - If no file named "filename" exists, a BIOFile
                           with name filename is created and opened for
                           read and write access. If creating and opening
                           are successful, the procedure returns
                           TRUE, otherwise FALSE.
            IsNew:       - TRUE if the BIOFile has been new created
                           (this supposes that ReadOnly = FALSE and that no
                           file named "filename" existed).
                         - FALSE if an existing BIOFILE has been opened.
            f:           When OpenBIOFile returns FALSE (see above), f remains
                         unchanged.
                         When OpenBIOFile returns TRUE (see above), the fields
                         of f contain:
                         - BIOFId:       a unique identifier of the BIOFile.
                         - SpecialBlock: - If IsNew = TRUE, SpecialBlock contains
                                           garbage
                                         - If IsNew = FALSE, SpecialBlock contains
                                           what the programmer had written into it
                                           before closing the BIOFile the last time.
                                         NOTE: The SpecialBlock may be used to
                                               store information about the BIOFile.
                                               Its usage is under the sole
                                               programmer's responsibility.
 RETURNS:   See "ReadOnly" above.
*)



PROCEDURE PutSegment(BIOFId      : BIOFileId;
                     NrOfBlocks  : CARDINAL;
                     SegmId      : SegmentId;
                     VAR Segment : ARRAY OF Block) : BOOLEAN;
(*
 NOTE:      Segment is a VAR parameter for efficiency reasons only.
```

```
 FUNCTION:  Initiate writing NrOfBlocks blocks identified by SegmId.
 PARAMETER: BIOFId:      BIOFileId of the BIOFile on which the segment
                         shall be written.
            NrOfBlocks: Number of blocks of the segment.
                         Must be less than or equal to the NrOfBlocks
                         declared in AllocateSegment and less than or
                         equal to HIGH(Segment) + 1
            SegmId:     Client's address of the segment
            Segment:    Segment to be written
 RETURNS:   TRUE if successful, FALSE otherwise.
*)


PROCEDURE PutSpecialBlock(VAR f : BIOFile) : BOOLEAN;
(*
 FUNCTION:  write f's SpecialBlock BIOFile into the external BIOFile.
 PARAMETER: f.BIOFId :       BIOFileId of the BIOFile whose SpecialBlock
                             shall be written
            f.SpecialBlock: Block to be written
 RETURNS:   TRUE if successful, FALSE otherwise.
*)


PROCEDURE RemoveBIOFile(filename : ARRAY OF CHAR) : BOOLEAN;
(*
 FUNCTION:  Remove the file with name "filename" if it exists and is a BIOFile.
 PARAMETER: filename:  name of the external BIOFile to remove.
 RETURNS:   TRUE if a file named "filename" existed, was a BIOFile
            and could be removed.
            FALSE otherwise.
*)


PROCEDURE WaitBIO(BIOFId  : BIOFileId) : BOOLEAN;
(*
 FUNCTION:   Awaits completion of any previously initiated I/O operation
             (* PutSegment or GetSegment) on the BIOFile with the
             specified BIOFileId.
 PARAMETER:  BIOFId: the BIOFileId of the BIOFile on which the performance
                     of a previously initiated I/O operation is to wait for.
 RETURNS:    TRUE if the previously initiated I/O operation is completed
             successfully or if there was no I/O operation initiated.
             FALSE otherwise.
 NOTE:       FALSE indicates an error occured during the execution of
             the previously initiated I/O operation.
 )


END HEsegmentIO.
```

```
DEFINITION MODULE HConstypes;
(* HConstypes ++ implementation dependant constants and types file & string *)
(* $HOST-COMPUTER:  SUN (other versions exist for Lilith and Macintosh)
   $AUTHOR:         Alfred Ultsch, Michel Kiener
   $DATE:           February 17th, 1987        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HConstypes.def
   $MODIFICATIONS:
   $DESCRIPTION:    This module defines constants and basic types depending
                    on the host computer.
*)

(* Import List: *)
(* ------------ *)
IMPORT SYSTEM; (* to import the type BYTE *)
FROM HChidden IMPORT hiddenfile;

(* Export List: *)
(* ------------ *)
EXPORT QUALIFIED
(* Constants: *)
(* constant indicating failure *)
 FAILED,          (* indicates failure                                    *)

(* constants for strings *)
 ENDSTR,          (* indicates the end of a string                        *)
 MAXSTR,          (* maximal number of CHAR's in one string               *)
 NEWLINE,         (* end-of-line character                                *)

 (* constants for files *)
 ENDFILE,  (* all read procedures return this when end of file is reached *)
 EOT,             (* this character signals end of file on a terminal     *)

 (* constants for the commandline interface *)
 BLANK,           (* definition of a spacing char (separates arguments)   *)
 ESC,             (* escape character to escape from execution of a progr *)
 OPTION,          (* a character to designate options on the commandline  *)
 TAB,             (* definition of a spacing char (separates arguments)   *)
 BS,              (* backspace one character                              *)
 DEL,             (* delete last input character                          *)

 (* constants for number ranges *)
 MAXCARD,         (* largest CARDINAL number                              *)
 MAXINT,          (* largest INTEGER number                               *)
 MININT,          (* smallest INTEGER number                              *)
 MAXREAL,         (* largest REAL number                                  *)
 MINREAL,         (* smallest REAL number                                 *)
 LEASTREAL,       (* -LeastReal is the maximum negative real              *)
 REALPRECISION,   (* below this constant REALs are equal                  *)

 (* constants for storage alignments *)
 BITSPERBYTE,        (* number of bits in one byte                        *)
 MOSTSIG,            (* place of most significant bit in a byte           *)
 LEASTSIG,           (* place of least significant bit in a word          *)
 BYTESPERWORD,       (* size in bytes of the standard type WORD           *)
 BYTESPERCARD,       (* size in bytes of the standard type CARDINAL       *)
 BYTESPERINT,        (* size in bytes of the standard type INTEGER        *)
 BYTESPERREAL,       (* size in bytes of the standard type REAL           *)
 BYTESPERBITSET,     (* size in bytes of the standard type BITSET         *)
 BYTESPERADDRESS,    (* size in bytes of the type ADDRESS                 *)
 BYTESPERSIZE,       (* if x is the size of a type returned by standard
                        function SIZE or TSIZE then (x * BYTESPERSIZE) is
```

```
                        the size in bytes of that type *)
 BYTESPERBLOCK,     (* length in BYTEs of blocks of storage used by HEsegmentIO *)


(* Types:  *)
 string,   (* string of the maximal possible length *)
 file,     (* Operations on this type are defined in the module HPfiles.
              Files have one of five IOmodes:
              IOREAD, IOWRITE, IOAPPEND, IOVERRIDE, IFTHERE.
              For a desciption see module HPfiles *)
 IOmode,   (* the different file opening modes *)
 BYTE;     (* 8-bit byte *)



(* Declaration of exported Constants: *)
(* ---------------------------------- *)
CONST
  FAILED              = -1;
  EOL                 = 12C;
  BITSPERBYTE         = 8;
  BLANK               = ' ';
  BS                  = 010C;
  BYTESPERBITSET      = 4;
  BYTESPERCARD        = 4;
  BYTESPERINT         = 4;
  BYTESPERREAL        = 4;
  BYTESPERWORD        = 4;
  BYTESPERADDRESS     = 4;
  BYTESPERBLOCK       = 256;
  BYTESPERSIZE        = 4;
  DEL                 = 177C;
  ENDFILE             = -1 ;
  ENDSTR              = 0C ;
  EOT                 = 04C;
  ESC                 = 033C;
  LEASTREAL           = 1.4694738E-39;
  LEASTSIG            = 31;
  MAXCARD             = 4294967295;
  MAXINT              = 2147483647;
  MAXREAL             = 3.4028E38;
  MAXSTR              = 256;
  MININT              = - 1 - MAXINT;
  MINREAL             = - MAXREAL;
  MOSTSIG             = 0;
  NEWLINE             = EOL;
  OPTION              = '-';
  REALPRECISION       = 1.0E-06;
  TAB                 = 011C;

(* Declaration of exported Types: *)
(* ------------------------------ *)
TYPE
  BYTE    = SYSTEM.BYTE; (* on Lilith: BYTE = CHAR; *)
  file    = RECORD
              filename : ARRAY [0..79] OF CHAR;
              (* Each variable of type "file" has a "filename"-field.
                 When a file is opened, the actual "name" is copied
                 into the "filename"-field of the file variable f, so that
                 it can be asked later. *)
              hf : hiddenfile;
```

```
        END;
  IOmode = (IOREAD, IOWRITE, IOAPPEND, IOVERRIDE, IFTHERE);
  string =  ARRAY [0..MAXSTR-1] OF CHAR;

END HConstypes.
```

APPENDIX B: DEFINITION MODULES OF THE UTILITIES DELIVERED WITH HOST

CONTENTS:

```
DEFINITION MODULE HUrwns;
(* HUrwns ++ read / write numbers and strings, no REAL type *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHOR:         Michel Kiener
   $DATE:           February 17th, 1987        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HUrwns.def
   $MODIFICATIONS:
   $DESCRIPTION:    Procedures to read numbers (no REAL type) and strings from
                    STDIN and from a file;
                    Procedures to write numbers in different formats (no REAL
                    type) and strings on STDOUT and on a file.
                    The functional difference between a Get...-procedure and
                    the corresponding Read...-procedure is that the
                    Get...-procedure does NOT distinguish (i.e. returns the
                    same value zero) betwween input zero and syntactically
                    incorrect input or input outside the legal range for that
                    type, but the Read...-procedure does.
                    The procedures to read from / write to a file have
                    the same names as the corresponding procedures to read
                    from STDIN / write to STDOUT, excepted that a letter "f"
                    is appended to their name. They have the same function,
                    parameters and return value than the procedures without
                    "f", excepted that they have an additional parameter
                    which is the file to read from / to write to.
   $NECESSARY CONDITIONS:  The ...f-procedures assume that the file
                    from which is read / on which is written to have been
                    properly opened through a call of HPfiles.open and will be
                    properly closed through a call of HPfiles.close.

Alphabetical List of Procedures:
--------------------------------
GetCard       -- read a CARDINAL as a string from STDIN
GetChar       -- read a CHAR from STDIN
GetInt        -- read an INTEGER as a string from STDIN
GetLine       -- read at most one line from STDIN
ReadCard      -- read a CARDINAL as a string from STDIN
ReadInt       -- read an INTEGER as a string from STDIN
WriteBin      -- write binary representation of a CARDINAL on STDOUT
WriteCard     -- write a CARDINAL as a string on STDOUT
WriteChar     -- write a CHAR on STDOUT
WriteHex      -- write hexadecimal representation of a CARDINAL on STDOUT
WriteInt      -- write an INTEGER as a string on STDOUT
WriteLn       -- write NEWLINE on STDOUT
WriteOct      -- write octal representation of a CARDINAL on STDOUT
WriteString   -- write a string on STDOUT

GetCardf      -- read a CARDINAL as a string from a file
GetCharf      -- read a CHAR from a file
GetIntf       -- read an INTEGER as a string from a file
GetLinef      -- read at most one line from a file
ReadCardf     -- read a CARDINAL as a string from a file
ReadIntf      -- read an INTEGER as a string from a file
WriteBinf     -- write binary representation of a CARDINAL on a file
WriteCardf    -- write a CARDINAL as a string on a file
WriteCharf    -- write a CHAR on a file
WriteHexf     -- write hexadecimal representation of a CARDINAL on a file
WriteIntf     -- write an INTEGER as a string on a file
WriteLnf      -- write NEWLINE on a file
WriteOctf     -- write octal representation of a CARDINAL on a file
WriteStringf  -- write a string on a file
*)
```

```
(********** IMPORTED OBJECTS:**************************)

FROM HConstypes     IMPORT (* type  *) file;

(*****************************************************)

EXPORT QUALIFIED
(* proc  *) GetCard,   GetChar,    GetInt,     GetLine,
            ReadCard,  ReadInt,
            WriteBin,  WriteCard,  WriteChar,  WriteHex,  WriteInt,  WriteLn,
            WriteOct,  WriteString,

            GetCardf,  GetCharf,   GetIntf,    GetLinef,
            ReadCardf, ReadIntf,
            WriteBinf, WriteCardf, WriteCharf, WriteHexf, WriteIntf,WriteLnf,
            WriteOctf, WriteStringf;


(*
Description of Procedures:
--------------------------
*)

PROCEDURE GetCard (VAR c : CARDINAL) : CARDINAL;
(*
 FUNCTION : Read characters from STDIN skipping leading BLANKs and TABs until
            - either a sequence of digits followed by a non-digit
            - or a sequence of non-digits followed by a digit or a NEWLINE
            is encountered.
            If the last read character is not a NEWLINE, put it back to
            STDIN (so that it can be read again by a following call of a
            Get...-procedure).
            The DEL character has the commonly known effect.
 PARAMETER: c becomes either the value 0 or the value of the read cardinal
            number.
            c becomes the value 0 when
             - either the read CARDINAL is 0
             - or the read digits build a cardinal number greater than MAXCARD
               (this is especially the case if the read sequence of digits has
               more digits than MAXCARD has; notice that the procedure goes on
               reading as many digits as encountered and stops only when it
               meets the next non-digit!)
             - or a sequence of non-digits followed by a digit or a NEWLINE
               has been read
 RETURNS  : the same value as the parameter c has.
*)


PROCEDURE GetChar (VAR ch : CHAR) : CHAR;
(*
 FUNCTION : read one character from STDIN
 PARAMETER: ch is the read char or EOT (which indicates the end of the
            STDIN-input)
 RETURNS  : the read CHAR (has the same value as ch), or EOT (which indicates
            the end of the STDIN-input)
*)


PROCEDURE GetInt (VAR i : INTEGER) : INTEGER;
```

```
(*
  FUNCTION : Read characters from STDIN skipping leading BLANKs and TABs until
             - either a sequence of digits preceeded by at most one minus
               sign and followed by a non-digit
             - or a sequence of non-digits followed by a digit or a minus sign
               or a NEWLINE is encountered.
             If the last read character is not a NEWLINE, put it back to
             STDIN (so that it can be read again by a following call of a
             Get...-procedure).
             The DEL character has the commonly known effect.
 PARAMETER: i becomes either the value 0 or the value of the read integer
             number.
             i becomes the value 0 when
              - either the read INTEGER is 0
              - or the read digits build an integer number greater than MAXINT
                or less than -1-MAXINT (this is especially the case if the read
                sequence of digits has more digits than MAXINT or -1-MAXINT
                have;
                notice that the procedure goes on reading as many digits as
                encountered and stops only when it meets the next non-digit!)
              - or a sequence of non-digits followed by a digit or a minus
                sign or a NEWLINE has been read
 RETURNS  : the same value as the parameter i has.
 CAUTION  : only numbers >= -1-MAXINT and <= MAXINT are translated into an
             INTEGER; numbers smaller than -1-MAXINT cannot be translated. By
             luck, it can be assumed that MININT = -1-MAXINT on a reasonnable
             computer!
*)

PROCEDURE GetLine (VAR line : ARRAY OF CHAR; max : INTEGER) : BOOLEAN;
(*
 FUNCTION : Read at most one line from STDIN.
             DEL and <CRTL> are proceeded.
 PARAMETER: line: place where the read line is stored to.
             max: the line is truncated to this length if desired.
 RETURNS  : true if a line is successfully obtained; FALSE implies end-of-file
*)


PROCEDURE ReadCard (VAR c : CARDINAL) : BOOLEAN;
(*
 FUNCTION : Read characters from STDIN skipping leading BLANKs and TABs until
             - either a sequence of digits followed by a non-digit
             - or a sequence of non-digits followed by either a digit or a
               NEWLINE
             is encountered.
             If the last read character is not a NEWLINE, put it back to
             STDIN (so that it can be read again by a following call of a
             Get...-procedure).
             The DEL character has the commonly known effect.
 PARAMETER: c becomes either the value 0 or the value MAXCARD or the value of
             the read cardinal number.
             c becomes the value 0 when
              - either the read cardinal number is 0
              - or a sequence of non-digits which does not contain only BLANKs
                or TABs and is followed by either a digit or a NEWLINE has
                been read
             c becomes the value MAXCARD when
             - either the read cardinal number is MAXCARD
```

```
            - or the read digits build a cardinal number greater than MAXCARD
              (this is especially the case if the read sequence of digits has
              more digits than MAXCARD has; notice that the procedure goes on
              reading as many digits as encountered and stops only when it
              meets the next non-digit!).
            Otherwise, c becomes the value of the read cardinal number.
 RETURNS  : true iff a cardinal number not greater than MAXCARD has been read.
*)


PROCEDURE ReadInt (VAR i : INTEGER) : BOOLEAN;
(*
  FUNCTION : Read characters from STDIN skipping leading BLANKs and TABs until
             - either a sequence of digits preceeded by at most one minus
               sign and followed by a non-digit
             - or a sequence of non-digits followed by a digit or a minus sign
               or a NEWLINE is encountered.
             If the last read character is not a NEWLINE, put it back to
             STDIN (so that it can be read again by a following call of a
             Get...-procedure).
             The DEL character has the commonly known effect.
  PARAMETER: i becomes either the value 0 or the value -1-MAXINT or the value
             MAXINT or the value of the read integer number.
             i becomes the value 0 when
              - either the read integer number is 0
              - or a sequence of non-digits followed by a digit or a minus
                sign or a NEWLINE has been read
             i becomes the value -1-MAXINT when
              - either the read integer number is -1-MAXINT
              - or the read digits build an integer number less than -1-MAXINT
                (this is especially the case if the read sequence of digits has
                more digits than -1-MAXINT has; notice that the procedure goes
                on reading as many digits as encountered and stops only when
                it meets the next non-digit!)
             i becomes the value MAXINT when
              - either the read integer number has the value MAXINT
              - or the read digits build an integer number greater than MAXINT
                (this is especially the case if the read sequence of digits has
                more digits than MAXINT has; notice that the procedure goes on
                reading as many digits as encountered and stops only when it
                meets the next non-digit!)
             Otherwise, i becomes the value of the read integer number.
  RETURNS  : true iff an integer number not less than -1-yMAXINT and not
             greater than MAXINT has been read.
  CAUTION  : only numbers >= -1-MAXINT and <= MAXINT are translated into an
             INTEGER; numbers smaller than -1-MAXINT cannot be translated. By
             luck, it can be assumed that MININT = -1-MAXINT on a reasonnable
             computer!
*)


PROCEDURE WriteBin(c, n : CARDINAL);
(*
  FUNCTION : Write on STDOUT the string which is the binary representation of
             the CARDINAL c with at least n characters.
             If n is smaller than the number of digits needed, n is ignored
             and the number of digits needed is written.
             If n is greater than the number of digits needed, blanks
             are added preceeding the number.
```

```
*)


PROCEDURE WriteCard(c, n : CARDINAL);
(*
 FUNCTION : Write on STDOUT the CARDINAL c as a string with at least
            n characters.
            If n is smaller than the number of digits needed, n is ignored
            and the number of digits needed is written.
            If n is greater than the number of digits needed, blanks
            are added proceeding the number.
*)


PROCEDURE WriteChar(ch : CHAR);
(*
 FUNCTION : Write the character ch on STDOUT.
*)


PROCEDURE WriteHex(c, n : CARDINAL);
(*
 FUNCTION : Write on STDOUT the string which is the hexadecimal
            representation of the CARDINAL c with at least n characters.
            If n is smaller than the number of digits needed, n is ignored
            and the number of digits needed is written.
            If n is greater than the number of digits needed, blanks
            are added preceeding the number.
*)


PROCEDURE WriteInt(i : INTEGER; n : CARDINAL);
(*
 FUNCTION : Write on STDOUT the INTEGER i as a string with at least
            n characters.
            If n is smaller than the number of digits needed, n is ignored
            and the number of digits needed is written.
            If n is greater than the number of digits needed, blanks
            are added preceeding the number.
*)


PROCEDURE WriteLn;
(*
 FUNCTION : Write the character NEWLINE on STDOUT.
*)


PROCEDURE WriteOct(c, n : CARDINAL);
(*
 FUNCTION : Write on STDOUT the string which is the octal representation of
            the CARDINAL c with at least n characters.
            If n is smaller than the number of digits needed, n is ignored
            and the number of digits needed is written.
            If n is greater than the number of digits needed, blanks
            are added preceeding the number.
*)


PROCEDURE WriteString(str : ARRAY OF CHAR);
```

```
(*
  FUNCTION : Write the string str on STDOUT.
*)


PROCEDURE GetCardf (VAR c : CARDINAL; VAR f : file) : CARDINAL;

PROCEDURE GetCharf (VAR ch : CHAR; VAR f : file) : CHAR;
(*
 FUNCTION : read one character from file f
 PARAMETER: ch is the read char or FS (imported from HConstypes, indicates
            end of file)
 RETURNS  : the read CHAR (has the same value as ch), or FS (imported from
            HConstypes, indicates end of file)
*)

PROCEDURE GetIntf (VAR i : INTEGER; VAR f : file) : INTEGER;
PROCEDURE GetLinef(VAR line:ARRAY OF CHAR; max :INTEGER; VAR f :file):BOOLEAN;
PROCEDURE ReadCardf (VAR c : CARDINAL; VAR f : file) : BOOLEAN;
PROCEDURE ReadIntf (VAR i : INTEGER; VAR f : file) : BOOLEAN;
PROCEDURE WriteBinf(c, n : CARDINAL; VAR f : file);
PROCEDURE WriteCardf(c, n : CARDINAL; VAR f : file);
PROCEDURE WriteCharf(ch : CHAR; VAR f : file);
PROCEDURE WriteHexf(c, n : CARDINAL; VAR f : file);
PROCEDURE WriteIntf(i : INTEGER; n : CARDINAL; VAR f : file);
PROCEDURE WriteLnf(VAR f : file);
PROCEDURE WriteOctf(c, n : CARDINAL; VAR f : file);
PROCEDURE WriteStringf(str : ARRAY OF CHAR; VAR f : file);

END HUrwns.
```

```
DEFINITION MODULE HUreal;
(* HUreal ++ read / write REAL numbers *)
(* $HOST-COMPUTER:  SUN, Lilith, Macintosh
   $AUTHOR:         Michel Kiener
   $DATE:           February 17th, 1987        $VERSION:    2.0
   $PROJECT:        HOST                       $FILE:       HUreal.def
   $MODIFICATIONS:
   $DESCRIPTION:    Procedures to read REAL numbers from STDIN and from a
                    file; procedures to write REAL numbers on STDOUT and on
                    a file.
                    The procedures to read from / write to a file have
                    the same names than the corresponding procedures to read
                    from STDIN / write to STDOUT, excepted that a letter "f"
                    is appended to their name. They have the same function,
                    parameters and return value than the procedures without
                    "f", excepted that they have an additional parameter
                    which is the file to read from / to write to.
   $NECESSARY CONDITIONS:  The ...f-procedures assume that the file
                    from which is read / on which is written to have been
                    properly opened through a call of HPfiles.open and will be
                    properly closed through a call of HPfiles.close.

Alphabetical List of Procedures:
--------------------------------
ReadReal      -- read a REAL as a string from STDIN
ReadRealf     -- read a REAL as a string from a file
WriteReal     -- write a REAL as a string on STDOUT
WriteRealf    -- write a REAL as a string on a file
*)


FROM HConstypes     IMPORT (* type  *) file;

EXPORT QUALIFIED (* proc  *) ReadReal, ReadRealf, WriteReal, WriteRealf;

(* Description of Procedures: *)
(* -------------------------- *)

PROCEDURE ReadReal(VAR r: REAL): BOOLEAN;
(*
 FUNCTION : Read characters from STDIN until a NEWLINE is encountered.
            Convert if possible the read string into a REAL.
            Format is the same as for the procedure HEreal.StringToReal.
 PARAMETER: r : the read real
 RETURNS  : FALSE if no syntactically valid number is found or if the number
            found is outside the machine defined range for the type REAL. TRUE
            otherwise.

*)


PROCEDURE ReadRealf (VAR r : REAL; VAR f : file) : BOOLEAN;
(*
 FUNCTION : Same as ReadReal, excpeted that the characters are read from the
            file f.
 RETURNS  : See ReadReal.
*)


PROCEDURE WriteReal(r:          REAL;
                    length,
```

```
                    precision:   CARDINAL;
                    scientific:  BOOLEAN);
(*
 FUNCTION : Convert the REAL r to a a string representing it and write it
            on STDOUT.
            Meaning of parameters and format are the same as in procedure
            HEreal.RealToString.
 PARAMETERS: See HEreal.RealToString.

*)


PROCEDURE WriteRealf(r:           REAL;
                     Field,
                     Digits:      CARDINAL;
                     scientific:  BOOLEAN;
                     VAR f:       file);
(*
 FUNCTION : Same as WriteReal, excpeted that the characters are written on
            file f.
*)


END HUreal.
```